A Dependently-typed Intermediate Language with General Recursion

Foo Bar Baz
The University of Foo
{foo,bar,baz}@foo.edu

Abstract

This is gonna to be written later.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Design

Keywords Dependent types, Intermediate language

1. Introduction

These are definitely drafts and only some main points are listed in each section.

a) Motivations:

- Because of the reluctance to introduce dependent types¹, the
 current intermediate language of Haskell, namely System
 F_C [11], separates expressions as terms, types and kinds,
 which brings complexity to the implementation as well as
 further extensions [13, 14].
- Popular full-spectrum dependently typed languages, like Agda, Coq, Idris, have to ensure the termination of functions for the decidability of proofs. No general recursion and the limitation of enforcing termination checking make such languages impractical for general-purpose programming.
- We would like to introduce a simple and compiler-friendly dependently typed core language with only one hierarchy, which supports general recursion at the same time.

b) Contribution:

- A core language based on Calculus of Constructions (CoC) that collapses terms, types and kinds into the same hierarchy.
- General recursion by introducing recursive types for both terms and types by the same μ primitive.

- Decidable type checking and managed type-level computation by replacing implicit conversion rule of CoC with generalized fold/unfold semantics.
- First-class equality by coercion, which is used for encoding GADTs or newtypes without runtime overhead.
- Surface language that supports datatypes, pattern matching and other language extensions for Haskell, and can be encoded into the core language.

c) Related work:

- Henk [5] and one of its implementation [7] show the simplicity of the Pure Type System (PTS). [8] also tries to combine recursion with PTS.
- Zombie [2, 9] is a language with two fragments supporting logics with non-termination. It limits the β-reduction for congruence closure [10].
- ΠΣ [1] is a simple, dependently-typed core language for expressing high-level constructions². UHC compiler [6] tries to use a simplified core language with coercion to encode GADTs.
- System F_C [11] has been extended with type promotion [14] and kind equality [13]. The latter one introduces a limited form of dependent types into the system³, which mixes up types and kinds.

2. Overview

BRUNO: Jeremy: can you give this section a go and start writing it up? I think this section should be your priority for now.

We begin this section with an informal introduction to the main features of λC_{β} . We show how it can serve as a simple and compiler-friendly core language with general recursion and decidable type system. The formal details are presented in Section 3.

2.1 Explicit Reduction Rules

BRUNO: Contrast our calculus with the calculus of constructions. Explain fold/unfold.

 λC_{β} is based on the *Calculus of Constructions* (λC) [4]. In contrast to the implicit reduction rules of λC , λC_{β} makes it explicit as to when and where to apply reduction rules.

Figure 1 is the so-called *conversion* rule of λC , which allows one to drive x:A from the derivation of x:B and the beta-equality of A and B. Note that in λC , the use of this rule is implicit

 $[Copyright\ notice\ will\ appear\ here\ once\ 'preprint'\ option\ is\ removed.]$

2015/5/16

¹ This might be changed in the near future. See https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase1.

² But the paper didn't give any meta-theories about the langauge.

³ Richard A. Eisenberg is going to implement kind equality [13] into GHC. The implementation is proposed at https://phabricator.haskell.org/D808 and related paper is at http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities-extended.pdf.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : s \qquad A =_{\beta} B}{\Gamma \vdash a : B}$$

Figure 1. The conversion rule of λC

in that it is automatically applied during type checking to all non-normal form terms. λC_{β} however differs in the following respects: 1) it eliminates the need to have the conversion rule; 2) it makes type conversion explicit by introducing two operations: cast[↑] and cast_↓.

In order to have a better intuition of the explicit reduction rules, let us consider a simple example. Suppose we have a built-in base type Int and

$$f \equiv \lambda x : (\lambda y : \star . y) \operatorname{Int}. x$$

Without the conversion rule, f cannot be applied to, say 3 in λC . Given that f is actually β -convertible to λx : Int. x, the conversion rule would allow the application of f to f. However in f0 is intended as an ill-typed application. Instead one would like to write the application as

$$f(\mathsf{cast}^{\uparrow}[(\lambda y: \star. y) \mathsf{Int}] 3)$$

The intuition is that, cast^\uparrow is actually doing type conversion since the type of 3 is Int and $(\lambda y : \star . y)$ Int can be reduced to Int.

The dual operation of cast^\uparrow is cast_\downarrow . The use of cast_\downarrow is better explained by another similar example. Suppose that

$$q \equiv \lambda x : \text{Int. } x$$

and z has type

$$(\lambda y:\star.y)$$
 Int

 $g\,z$ is again an ill-typed application, while $g\,(\mathsf{cast}_\downarrow\,z)$ is type correct because cast_\downarrow reduces the type of z to Int.

2.2 Decidability and Strong Normalization

BRUNO: Informally explain that with explicit fold/unfold rules the decidability of the type system does not depend on strong normalization

The decidability of the type system of λC depends on the normalization property for all constructed terms [3]. However strong normalization does not hold with general recursion. This is simply because due to the conversion rule, any non-terminating term would force the type checker to go into an infinitely loop, thus rendering the type system undecidable.

With explicit reduction rules, however, the decidability of the type system no longer depends on the normalization property. In fact λC_{β} is not strong normalizing, as we will see in later sections. The ability to write non-terminating terms forces us to have more control over type-level computation. To illustrate, let us consider a contrived example. Suppose that d is a "dependent type" where

$$d: \mathsf{Int} \to \star$$

so that $d\,3$ or $d\,100$ all yield the same type. With general recursion at hand, we can image a term z that has type

$$d\left(\operatorname{fix}\left(\lambda y:\operatorname{Int.}y\right)\right)$$

Apparently evaluating $fix(\lambda y : Int. y)$ would give us an infinite evaluation sequence, always yielding the same term. What would happen if we try to type check the following application:

$$(\lambda x : d \cdot 3.x)z$$

Under the normal typing rules of λC , the type checker would get stuck as it tries to do β -equality on two terms: d 3 and fix (λy : Int. y), where the latter is non-terminating.

This is not the case for λC_{β} : 1) it has no such conversion rule, therefore the type checker would do syntactic comparison between

the two terms instead of β -equality in the above example; 2) one would need to write infinitely $\mathsf{cast}^\uparrow / \mathsf{cast}_\downarrow$ to make the type checker loop forever (e.g., $(\lambda x:d3.x)$ ($\mathsf{cast}_\downarrow \, \mathsf{cast}_\downarrow \dots z$)). Apparently this is impossible in reality.

In summary, λC_{β} approaches the decidability of the type system by explicitly controlling type-level computation, which is independent of the normalization property, while supporting general recursion at the same time.

2.3 Unifying Recursive Types and Recursion

BRUNO: Show how in λC_{β} recursion and recursive types are unified. Discuss that due to this unification the sensible choice for the evaluation strategy is call-by-name.

2.4 Encoding Datatypes

BRUNO: Informally explain how to encode recursive datatypes and recursive functions using datatypes.

3. The Explicit Calculus of Constructions

BRUNO: Linus: can you write up this section? I think this section should be your priority. First bring in all results and formalization: syntax; semantics; proofs ... then write text

This section formalizes the syntax and semantics of the explicit calculus of constructions. This section also shows that how in the explicit calculus of constructions decidability of the type system does not depend on strong normalization.

- Give an overview of the core language and its syntax.
- Show the typing rules and operational semantics.
- The original formalization is suggested to rewrite using ott⁴ which is a standard in academia. For example, the formalization of GHC https://github.com/ghc/ghc/tree/master/docs/core-spec.
- Give formal proof of the soundness of the core language.
- Subject reduction and progress theorems will be proved.

4. The Explicit Calculus of Constructions with Recursion

BRUNO: Linus and Jeremy, I think you should do this section together. Most work is on Linus though since he needs to work out the proofs. Jeremy is mostly for Linus to consult with here:).

This section shows how to extend $\lambda C\beta$ with recursion. This extension allows the calculus to account for both: 1) recursive definitions; 2) recursive types. The extension preserves the decidability and soundness of the type system.

5. Surface language

BRUNO: Jeremy, I think you should write up this section.

- Expand the core language with datatypes and pattern matching by encoding.
- Give translation rules.
- Encode GADTs and maybe other Haskell extensions? GADTs seems challenging, so perhaps some other examples would be datatypes like *Fixf*, and *Monad* as a record. Could formalize records in Haskell style.

2 2015/5/16

⁴http://www.cl.cam.ac.uk/~pes20/ott/

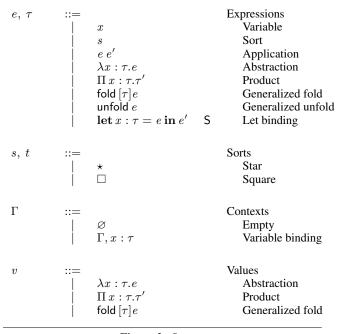


Figure 2. Syntax

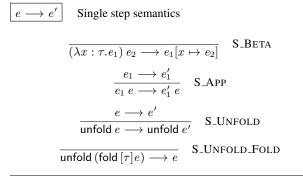


Figure 3. Dynamic semantics

6. Related Work

7. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

References

- T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. ΠΣ: Dependent types without the sugar. In *Functional and Logic Programming*, pages 40–55. Springer, 2010.
- [2] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. ACM SIGPLAN Notices, 49(1):33–45, 2014.
- [3] T. Coquand. Une théorie des constructions. PhD thesis, 1985.
- [4] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95-120, Feb. 1988. ISSN 0890-5401. URL http://dx.doi.org/10.1016/0890-5401(88)90005-3.
- [5] S. P. Jones and E. Meijer. Henk: a typed intermediate language. 1997.

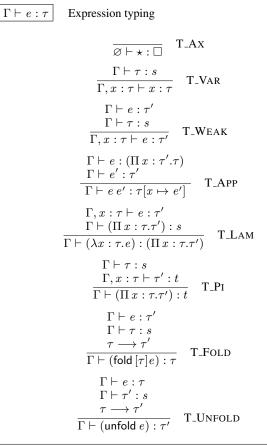


Figure 4. Typing rules

- [6] A. Middelkoop, A. Dijkstra, and S. D. Swierstra. A lean specification for gadts: system f with first-class equality proofs. *Higher-Order and Symbolic Computation*, 23(2):145–166, 2010.
- [7] J.-W. Roorda and J. Jeuring. Pure type systems for functional programming. 2007.
- [8] P. G. Severi and F.-J. J. de Vries. Pure type systems with corecursion on streams: from finite to infinitary normalisation. In ACM SIGPLAN Notices, volume 47, pages 141–152. ACM, 2012.
- [9] V. Sjöberg. A Dependently Typed Language with Nontermination. PhD thesis, University of Pennsylvania, 2015.
- [10] V. Sjöberg and S. Weirich. Programming up to congruence. In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pages 369–382, New York, NY, USA, 2015. ACM.
- [11] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.
- [12] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. *Typed compilation of recursive datatypes*, volume 38. ACM, 2003.
- [13] S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed haskell: System fc with kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Program*ming, ICFP, volume 13. Citeseer, 2013.
- [14] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

3 2015/5/16

A. Appendix Title

Additional proof goes here.

4 2015/5/16