DRAFT

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 langauge

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 types[1], 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].
- $\Pi\Sigma$ [1] is a simple, dependently-typed core language for expressing high-level constructions[2]. 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 system[3], 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

[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: $\mathsf{cast}^\uparrow$ and $\mathsf{cast}_\downarrow$.

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

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

Without the conversion rule, f cannot be applied to, say 3 in λC. Given that f is actually β-convertible to $\lambda x : \mathsf{Int}.\, x$, the conversion rule would allow the application of f to 3. However in λC_β, $f\,3$ 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, $\mathsf{cast}^\uparrow$ is actually doing type conversion since the type of 3 is Int and $(\lambda y : \star.\, y)\,\mathsf{Int}$ can be reduced to Int.

The dual operation of $\mathsf{cast}^\uparrow$ is $\mathsf{cast}_\downarrow$. The use of $\mathsf{cast}_\downarrow$ is better explained by another similar example. Suppose that

$$g \equiv \lambda x : \mathsf{Int}.\, x$$

and z has type

$$(\lambda y : \star.\, y)\,\mathsf{Int}$$

$g\,z$ is again an ill-typed application, while $g\,(\mathsf{cast}_\downarrow\,z)$ is type correct because $\mathsf{cast}_\downarrow$ 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

$$d\,(\mathsf{fix}\,(\lambda y : \mathsf{Int}.\, y))$$

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

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

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 $\mathsf{fix}\,(\lambda y : \mathsf{Int}.\, y)$, 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., $(\lambda x : d\,3.\, x)\,(\mathsf{cast}_\downarrow\,\mathsf{cast}_\downarrow\,\ldots z)$). 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.

Recursive types arise naturally if we want to do general recursion. λC_β differs from other programming languages in that it unifies both recursion and recursive types by the same μ primitive.

Recursive types. In the literature on type systems, there are two approaches to recursive types. One is called *equi-recursive*, the other *iso-recursive*. λC_β takes the latter approach since it is more intuitive to us with regard to recursion. The *iso-recusive* approach treats a recursive type and its unfolding as different, but isomorphic. In λC_β, this is witnessed by first $\mathsf{cast}^\uparrow$, then $\mathsf{cast}_\downarrow$. A classic example of recursive types is the so-called "hungry" type: $H = \mu\sigma : \star.\,\mathsf{Int} \to \sigma$. A function of type H can accept any number of numeric arguments and return a new function that is hungry for more, as illustrated below:

$$\mathsf{cast}_\downarrow\, H : \mathsf{Int} \to H$$
$$\mathsf{cast}_\downarrow\,\mathsf{cast}_\downarrow\, H : \mathsf{Int} \to \mathsf{Int} \to H$$
$$\mathsf{cast}_\downarrow\,\mathsf{cast}_\downarrow\,\ldots H : \mathsf{Int} \to \mathsf{Int} \to \cdots \to H$$

Recursion. The same μ primitive can also be used to define recursive functions, e.g., the factorial function:

$$\mu f : \mathsf{Int} \to \mathsf{Int}.\, \lambda x : \mathsf{Int}.\, \mathsf{if}\ (x == 0)\ \mathsf{then}\ 1\ \mathsf{else}\ x * f\,(x-1)$$

This is reflected by the dynamic semantics of the μ primitive:

$$\mu x : T.\, E \longrightarrow E[x := \mu x : T.\, E]$$

which is exactly doing recursive unfolding of the same term.

Due to the unification, the *call-by-value* strategy seems unfit to us. JEREMY: explain

2.4 Encoding Datatypes

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

With the explicit reduction rules and the μ primitive, it is straightforward to encode recursive datatypes and recusive functions using datatypes. While inductive datatypes can be encoded using either the Church or the Scott encoding, we adopt the Scott encoding as it is more fit with our unified recursive types. We demonstrate the encoding method using a simple datatype as a running example: the natural numbers.

Written in GADT-style, the datatype for natural numbers is:

$$\textbf{data}\ \mathsf{Nat} : \star\ \textbf{where}$$
$$\mathsf{zero} : \mathsf{Nat}$$
$$\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$$

In the Scoot encoding, the encoding of the Nat type reflects how its two constructors are going to be used. Since Nat is a recursive datatype, we have to use recursive types at some point to reflect its recursive nature. As it turns out, the Nat type can be simply represented as

$$\mu X : \star.\, \Pi b : \star.\, b \to (X \to b) \to b$$

As can be seen, in the function type $b \to (X \to b) \to b$, b corresponds to type of the zero constructor, and $X \to b$ corresponds to the type of the suc constructor. The intuition is that any use of the datatype being defined in the constructors is replaced with the recursive type, except for the return type, which is a type variable for use in the recursive functions.

Now its two constructors can be encoded correspondingly as below:

$$\textbf{let}\ \mathsf{zero} : \mathsf{Nat} = \mathsf{cast}^{\uparrow}[\mathsf{Nat}]\ (\lambda(b : \star)(z : b)(f : \mathsf{Nat} \to b).\ z)\ \textbf{in}$$
$$\textbf{let}\ \mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat} = \lambda(n : \mathsf{Nat}).\ \mathsf{cast}^{\uparrow}[\mathsf{Nat}]\ (\lambda(b : \star)(z : b)$$
$$(f : \mathsf{Nat} \to b).\ f\ n)\ \textbf{in}$$

Thanks to the explicit reduction rules, we can make use of the $\mathsf{cast}^{\uparrow}$ operation to do type conversion between the recursive type and its unfolding.

As the last example, let us see how we can define recursive functions using the Nat datatype. A simple example would be recursively adding two natural numbers, which can be defined as below:

$$\mu f : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}.\ \lambda n : \mathsf{Nat}.\ \lambda m : \mathsf{Nat}.$$
$$(\mathsf{cast}_{\downarrow}\ n)\ \mathsf{Nat}\ m\ (\lambda n' : \mathsf{Nat}.\ \mathsf{suc}\ (f\ n'\ m))$$

As we can see, the above definition quite resembles the case analysis common in modern functional programming languages. (Actually we formalize the encoding of case analysis as shown in Section 5.)

Due to the unification of recursive types and recursion, it facilitates encoding of recursive functions using recursive 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

In this section, we present a variant of the Calculus of Constructions (CC), called *explicit* Calculus of Constructions (ExpCC), which is the foundation of our core language λC_β. ExpCC can be regarded as λC_β without general recursion, so that has more straightforward properties and metatheory. It is suitable for illustrating the core idea of our design, that is to control β-reduction at the type level by introducing *explicit* type conversion semantics. This also brings a benefit to type checking of ExpCC, that the strong normalization is no long necessary to achieve the decidability of type checking. In the following part of this section, we give explanation of these properties by showing the syntax, static and dynamic semantics and the metatheory of ExpCC.

3.1 Syntax

The basic syntax of ExpCC is shown in Figure 2, which gives abstract syntax of expressions, sorts, contexts and values. Just like CC, ExpCC has two main advantages of keeping syntax concise when compared to the System F families including System F_ω and F_C. One is that ExpCC uses a single syntactic level to represent terms, types and kinds, which are usually distinguished in System F families. This brings the economy that we can use a single set of rules for terms, types and kinds uniformly. We use metavariables e and τ when referring to a "term" and a "type" respectively. Note that without distinction of terms, types and kinds, the "term" can be a term, a type or a kind. For example, in $\alpha : \star$, the "term" α is a type and the "type" of α is $\star$, which is a kind.

Another advantage is that ExpCC includes a product form $\Pi\, x : \tau_1 . \tau_2$ which is used to represent type of functions from values of type τ_1 to values of type τ_2. Compared with concepts in System F, $\Pi\, x : \tau_1 . \tau_2$ subsumes both the arrow of function types $\tau_1 \to \tau_2$ (if x does not occur free in τ_2), and the universal quantification $\forall x : \tau_1 . \tau_2$. Moreover, if x occurs free in τ_2, the product becomes a dependent product, which allows to represent dependent types. The product Π keeps the syntax of ExpCC simple and expressive at the same time.

The syntax difference of from CC is that ExpCC introduces two new explicit type conversion primitives, namely $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$ (pronounced as "cast up" and "cast down"), in order to replace the implicit conversion rule of CC. They represent two directions of type conversion operations: $\mathsf{cast}_{\downarrow}$ stands for the reduction of types while $\mathsf{cast}^{\uparrow}$ is the inverse. Specifically speaking, suppose we have $e : \sigma$, i.e. the type of expression e is σ. $\mathsf{cast}^{\uparrow}\ [\tau]\, e$ converts the type of e to τ, if there exists a type τ such that it can be reduced to σ in a single step, i.e. $\tau \longrightarrow \sigma$. $\mathsf{cast}_{\downarrow}\ e$ represents the one-step-reduced type of e, i.e. $(\mathsf{cast}_{\downarrow}\ e) : \sigma'$ if $\sigma \longrightarrow \sigma'$.

The intention of introducing two explicit cast primitives is that we can gain full control of computation at the type level by manually managing the type conversions. Later in §3.3 we will see dropping the implicit conversion rule of CC simplifies the type checking and leads to syntax-directed typing rules. This also influences the requirements of decidable type checking, that strong normalization is no long necessary.

$e,\ \tau$	$::=$		Expressions
	$\mid$	x	Variable
	$\mid$	s	Sort
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\lambda x : \tau . e$	Abstraction
	$\mid$	$\Pi\, x : \tau_1 . \tau_2$	Product
	$\mid$	$\mathsf{cast}^{\uparrow}\ [\tau]\, e$	Cast up to type
	$\mid$	$\mathsf{cast}_{\downarrow}\ e$	Cast down by reduction
$s,\ t$	$::=$		Sorts
	$\mid$	$\star$	Star
	$\mid$	$\square$	Square
Γ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Gamma, x : \tau$	Variable binding
v	$::=$		Values
	$\mid$	$\lambda x : \tau . e$	Abstraction
	$\mid$	$\Pi\, x : \tau_1 . \tau_2$	Product
	$\mid$	$\mathsf{cast}^{\uparrow}\ [\tau]\, e$	Cast up

Figure 2. Syntax of ExpCC

3.2 Syntactic sugar

LINUS: This part can be moved to the next section for λC_β.

To keep the core language minimal and simplify the translation of surface language, we use syntactic sugar shown in Figure 3 for ExpCC.

Let binding for $x = e_2$ in e_1 is equivalent to the substitution of x in e_1 with e_2, which can be reduced from $(\lambda x : \tau . e_1)\ e_2$.

The syntactic sugar for the function type is discussed in §3.1 for the functionality of the product Π. The product $\Pi\, x : \tau_1 . \tau_2$ can also be simply denoted by $\Pi_{\_} : \tau_1 . \tau_2$, where the underscore stands for an anonymous variable.

3.3 Type system

The type system for ExpCC contains typing judgements and operational semantics. Figure 4 lists operational semantics for ExpCC that defines rules for one-step reduction, including the β-reduction

Let binding	$\textbf{let}\, x : \tau = e_2 \,\textbf{in}\, e_1$	$\triangleq$	$(\lambda x : \tau . e_1)\ e_2$
Function type	$\tau_1 \rightarrow \tau_2$	$\triangleq$	$\Pi\, x : \tau_1 . \tau_2$ (x does not occur free in τ_2)

Figure 3. Syntatic sugar

rule and $\mathsf{cast}_{\downarrow}$ rules. The expressions will be reduced by applying rules one or more times. Rule S_CASTDOWN prevents the reduction from stalling with $\mathsf{cast}_{\downarrow}$ and continues to reduce the inner expression. Rule S_CASTDOWNUP states that $\mathsf{cast}_{\downarrow}$ cancels the $\mathsf{cast}^{\uparrow}$ of an expression.

$\boxed{e \longrightarrow e'}$ One-step reduction

$$\frac{}{(\lambda x : \tau . e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]}\ \textsc{S\_Beta}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \textsc{S\_App}$$

$$\frac{e \longrightarrow e'}{\mathsf{cast}_{\downarrow}\ e \longrightarrow \mathsf{cast}_{\downarrow}\ e'}\ \textsc{S\_CastDown}$$

$$\frac{}{\mathsf{cast}_{\downarrow}\ (\mathsf{cast}^{\uparrow}\ [\tau] e) \longrightarrow e}\ \textsc{S\_CastDownUp}$$

Figure 4. Operational semantics of ExpCC

Figure 5 lists the typing judgements to check the validity of expressions. Most rules are straightforward and similar with the ones in CC. For example, rule T_AX states that the "type" of sort $\star$ is a kind. This is derived from an axiom in CC, that the highest sort is $\Box$, making the type system predicative. Rule T_PI allows us to type dependent products. There are four possible combinations of types of τ_1 and τ_2 in a product $\Pi\, x : \tau_1 . \tau_2$, i.e. $(s, t) \in \{\star, \Box\} \times \{\star, \Box\}$. For some $(\lambda x : \tau_1 . e) : (\Pi\, x : \tau_1 . \tau_2)$, when $(s, t) = (\star, \Box)$, $x : \tau_1 : \star$, $e : \tau_2 : \Box$, so x is a term and e is a type. Thus, we have a type depending on a term which means the product is a dependent type.

The difference from CC for typing rules of ExpCC is that rule T_CASTUP and T_CASTDOWN are added to check the type conversion primitives $\mathsf{cast}^{\uparrow}$ and $\mathsf{cast}_{\downarrow}$, and the implicit type conversion rule of CC is removed, which is the rule as follows:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 =_{\beta} \tau_2}{\Gamma \vdash e : \tau_2}\ \textsc{Tcc\_Conv}$$

This rule is necessary for CC because of the premise requirements of the application rule T_APP:

$$\frac{\Gamma \vdash e_1 : (\Pi\, x : \tau_2 . \tau_1) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1[x \mapsto e_2]}\ \textsc{T\_App}$$

Consider the following two cases of the term $e_1\ e_2$:

- e_2 can be an arbitrary term so its type τ_2 is not necessary in normal form which might break the type checking of e_1, e.g. suppose $e_1 : \sigma \rightarrow \tau$ and $e_2 : \tau_2$, where τ_2 is an application $(\lambda x : \star . x)\ \sigma$. By TCC_CONV, $(\lambda x : \star . x)\ \sigma$ is β-equivalent to σ, thus $e_2 : \sigma$ and we can further use T_APP to achieve $e_1\ e_2 : \tau$.
- The type of e_1 should be a product expression according to the premise. But without the conversion rule, the term fails to type check if the type of e_1 is an expression which can further evaluate to a product, e.g. $\Pi y : ((\lambda x : \star . x)\, \tau_2) . \tau_1$. After applying TCC_CONV, the type of e_1 is converted to its β-equivalence $\Pi x : \tau_2 . \tau_1$. Thus we can further apply the T_APP.

We need to show that explicit type conversion rules with cast primitives can also satisfy the premises of rule T_APP. Still consider the above two cases:

- Given $e_1 : \sigma \rightarrow \tau$ and $e_2 : (\lambda x : \star . x)\ \sigma$, we do the application by term $e_1\ (\mathsf{cast}_{\downarrow}\ e_2)$. Since $(\lambda x : \star . x)\ \sigma \longrightarrow \sigma$, $\mathsf{cast}_{\downarrow}\ e_2 : \sigma$, the term $e_1\ (\mathsf{cast}_{\downarrow}\ e_2)$ type-checks with the rule T_APP.
- Given $e_1 : (\Pi y : ((\lambda x : \star . x)\, \tau_2) . \tau_1)$ and $e_2 : \tau_2$, we do the application by term $e_1\ (\mathsf{cast}^{\uparrow}\ [(\lambda x : \star . x)\, \tau_2] e_2)$. Noting that $(\lambda x : \star . x)\, \tau_2 \longrightarrow \tau_2$, the term conforms to rule T_CASTUP. Thus $\mathsf{cast}^{\uparrow}\ [(\lambda x : \star . x)\, \tau_2] e_2 : ((\lambda x : \star . x)\, \tau_2)$ and the term $e_1\ (\mathsf{cast}^{\uparrow}\ [(\lambda x : \star . x)\, \tau_2] e_2)$ can be type-checked by the rule T_APP.

Therefore, it is feasible to replace implicit conversion rules of CC with explicit type conversion rules.

$\boxed{\Gamma \vdash e : \tau}$ Expression typing

$$\frac{}{\varnothing \vdash \star : \Box}\ \textsc{T\_Ax}$$

$$\frac{\Gamma \vdash \tau : s}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{T\_Var}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : s}{\Gamma, x : \tau_1 \vdash e : \tau_2}\ \textsc{T\_Weak}$$

$$\frac{\Gamma \vdash e_1 : (\Pi\, x : \tau_2 . \tau_1) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1[x \mapsto e_2]}\ \textsc{T\_App}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash (\Pi\, x : \tau_1 . \tau_2) : s}{\Gamma \vdash (\lambda x : \tau_1 . e) : (\Pi\, x : \tau_1 . \tau_2)}\ \textsc{T\_Lam}$$

$$\frac{\Gamma \vdash \tau_1 : s \quad \Gamma, x : \tau_1 \vdash \tau_2 : t}{\Gamma \vdash (\Pi\, x : \tau_1 . \tau_2) : t}\ \textsc{T\_Pi}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : s \quad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}^{\uparrow}\ [\tau_1] e) : \tau_1}\ \textsc{T\_CastUp}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_2 : s \quad \tau_1 \longrightarrow \tau_2}{\Gamma \vdash (\mathsf{cast}_{\downarrow}\ e) : \tau_2}\ \textsc{T\_CastDown}$$

Figure 5. Typing rules of ExpCC

3.4 Decidability of type checking

The conversion rule of CC is not syntax-directed because it can be implicitly applied at any time in a derivation. The β-equality premise of the rule also leads to the decidability of type checking relying on the strong normalization property of CC:

Theorem 3.1 (Necessity of strong normalization)
The strong normalization property of CC is necessary for the decidability of its type checking.

Proof. The proof is by contradiction. Suppose strong normalization does not hold in the type system, then we can find a type τ_1 such that there exists at least one reduction sequence which does not terminate. Notice that any type τ_2 in such reduction sequence holds for $\tau_1 =_{\beta} \tau_2$. Thus we can constantly apply the conversion rule without termination and the type checking will not stop, which means the type checking is undecidable. $\Box$

Requiring strong normalization to achieve the decidability of type checking makes it impossible to combine general recursion with CC, because general recursion might cause nontermination which simply breaks the strong normalization property. So we use explicit type conversion rules by cast operations to relax the constraints of achieving decidable type checking. We have the following theorem:

Theorem 3.2 (Decidability of type chechking for ExpCC)
Let Γ be an environment, e and τ be expressions of ExpCC such that $\Gamma \vdash \tau : s$. Then the problem of knowing if one has $\Gamma \vdash e : \tau$ is decidable.

Proof. By induction on typing rules in Figure 5. □

Notice that new explicit type conversion rules are syntax-directed and do not include the β-equality premise but one-step reduction instead. Because checking if one term is one-step-reducible to the other is always decidable by enumerating the reduction rules, type checking using these rules are always decidable. Therefore the proof of decidability for ExpCC does not rely on the strong normalization. This also implies the possibility of introducing general recursion into the system with decidable type checking.

3.5 Soundness

LINUS: Is it necessary to show soundness in this section? How about merging with the core language? Because we have to show the soundness of λC_β in later sections anyway.

It is straightforward to obtain the soundness of ExpCC by combining the following two theorems:

Theorem 3.3 (Subject Reduction)
If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$.

Proof. By induction on rules in Figure 4. □

Theorem 3.4 (Progress)
If $\varnothing \vdash e : \tau$ then either e is a value v or there exists e' such that $e \longrightarrow e'$.

Proof. By induction on rules in Figure 5. □

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 $Fix f$, and $Monad$ as a record. Could formalize records in Haskell style.

6. Related Work

7. Conclusion

Conclusion and related work.

Acknowledgments

Thanks to Blah. This work is supported by Blah.

References

[1] 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.

[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 Programming, 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.

A. Appendix Title

Additional proof goes here.