

BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

SYSTEM F WITH CONSTRAINT TYPES

by

KEVIN DONNELLY

B.S., Carnegie Mellon University, 2004

Submitted in partial fulfillment of the
requirements for the degree of

Master of Arts

2008

Approved by

First Reader

Hongwei Xi, PhD
Associate Professor of Computer Science

Second Reader

Assaf Kfoury, PhD
Professor of Computer Science

SYSTEM F WITH CONSTRAINT TYPES

KEVIN DONNELLY

ABSTRACT

System F is a type system that can be seen as both a proof system for second-order propositional logic and as a polymorphic programming language. In this work we explore several extensions of System F by types which express subtyping constraints. These systems include terms which represent proofs of subtyping relationships between types. Given a proof that one type is a subtype of another, one may use a coercion term constructor to coerce terms from the first type to the second. The ability to manipulate type constraints as first-class entities gives these systems a lot of expressive power, including the ability to encode generalized algebraic data types and intensional type analysis. The main contributions of this work are in the formulation of constraint types and a proof of strong normalization for an extension of System F with constraint types.

Contents

1	Introduction	1
2	System F	3
2.1	Type-Dependent Operations in System F	5
2.2	Impredicative Encodings of Algebraic Datatypes	6
3	System F with Constraint Types	9
3.1	Definition of λ_{2C}	9
3.2	Impredicative Encodings of GADTs	13
3.3	Basic Meta-theory of λ_{2C}	19
4	Strong Normalization for λ_{2C^-}	27
5	$\lambda_{2C^-}^{(),\times,+}$: An Extension of λ_{2C^-}	43
5.1	Definition of $\lambda_{2C^-}^{(),\times,+}$	43
5.2	Strong Normalization for $\lambda_{2C^-}^{(),\times,+}$	46
6	Related Work	55
7	Conclusion	55

LIST OF ABBREVIATIONS

Abbreviation or Symbol	Definition
λ_{2C}	System F with full constraint types, introductions and eliminations
λ_{2C^-}	System F with constraint types and introductions
$\lambda_{2C^-}^{(),\times,+}$	λ_{2C^-} extended by unit, product and sum types with constraint types, introductions and eliminations
ATS	Applied Type System
GADT	Generalized Algebraic Data Type
GHC	Glasgow Haskell Compiler
GRDC	Guarded Recursive Data Type Constructor

1 Introduction

System F is an expressive type-system which corresponds, via the Curry-Howard isomorphism, to second-order propositional logic. The main uses of System F, with various extensions or restrictions, are as programming language type systems, type intermediate languages in compilers, and languages for mathematical definitions and proofs in theorem proving systems.

Modern functional languages like Standard ML [11], OCaml [10] and Haskell [9] can be seen a restriction of System F. The Glasgow Haskell Compiler (GHC) [4] uses a variant of System F as a typed intermediate language during the translation from source code to machine code. The Coq theorem proving system [2] is based on the Calculus of Constructions [3], which is an extension of System F. The ATS language [1], which is both a programming language and a theorem proving language, is similarly based on an extension of System F.

Many features of languages based on System F, in particular generalized algebraic datatypes [17] (GADTs) and intensional type analysis [8], are difficult to translate directly into System F, and hence have required various extensions in order to be expressed. A common example of a GADT is one used for well-typed terms of an object language

```
datatype TERM(type) =
  | TERMint(int) of int
  | TERMplus(int) of (TERM(int),TERM(int))
  | TERMleq(bool) of (TERM(int),TERM(int))
  | {a:type} TERMite(a) of (TERM(bool), TERM(a), TERM(a))
```

What distinguishes GADTs such as `TERM(type)` from normal algebraic datatypes is that the parameter to the datatype is different for each datatype constructor, which can allow us to infer constraints on type variables based on the data constructor used. For example if `TERMint(0) : TERM(a)` then it must be that `a = int`.

Intensional type analysis allows for so-called *ad-hoc* polymorphism, in which type parameters are inspected by polymorphic functions so the functions may behave different at different type arguments. One way for this to be implemented is to have terms which represent types be passed to such functions, which can then analyze these terms. For example, if `REP t` is a type for the representation of the type `t` then we may write a function

```
to_string : ∀ a:type. REP a -> a -> String
```

Such a type representation can be expressed a GADT:

```
datatype REP(type) =
  | REPunit(unit)
  | {t1:type, t2:type} REPfun(t1 -> t2) of (REP t1, REP t2)
  | {t1:type, t2:type} REPprod(t1 * t2) of (REP t1, REP t2)
  | {t1:type, t2:type} REPsum(t1 + t2) of (REP t1, REP t2)
```

It has been observed that the addition of type constraints to System F, along with the ability to use these constraints when type-checking terms, allows for the expression of GADTs and intensional type analysis. For example, the TERM example above can be expressed using constraints as:

```
datatype TERM(a:type) =
  | {a == int} TERMint of int
  | {a == int} TERMplus of (TERM(int),TERM(int))
  | {a == bool} TERMleq of (TERM(int),TERM(int))
  | TERMite of (TERM(bool), TERM(a), TERM(a))
```

A significant advantage of constraint-based formulations of GADTs is that it allows for relatively simple rules for typing case-expressions over GADTs. When typing each branch of the case, the constraints corresponding to constructor for that branch are added to the constraint context. For example, consider a case statement over $e : \text{TERM } a$. When typing the body of the branch for $\text{TERMint}(x)$, we can make use of the constraint $a = \text{int}$.

We have observed that one rather simple and flexible way of adding constraints to System F is to reify type constraints as types and derivations of type constraint satisfaction as terms. We introduce the concept of constraint types and formulate an extension of System F with constraint types. We prove that the extension with constraint types preserves type soundness, so it is reasonable to use constraint types in a programming language or typed intermediate language. We also show that, when certain restrictions are imposed, the extension of System F with constraint types is strongly normalizing. This shows that System F with constraint types can form the basis of a theorem proving system. We have found that adding constraint types alone to System F is not enough to encode most uses of GADTs. However, we show that with the addition of higher-order sorts, constraint types can allow for the encoding of GADTs and common functions on them.

The remainder of the thesis is structured as follows: First we informally review System F (Section 2), then we introduce constraint types and a formal calculus, λ_{2C} , with constraint types, we explore the encodings of GADTs and the limitations of such encodings, and we prove some basic properties of λ_{2C} , including type soundness (Section 3), then we define λ_{2C^-} , a restriction of λ_{2C} , and prove strong normalization (Section 4), then we extend λ_{2C^-} with unit, product and sum types, yielding $\lambda_{2C^-}^{(),\times,+}$, and prove strong normalization (Section 5), finally we discuss related work (Section 6) and conclude (Section 7).

2 System F

System F [6][5], also known as the second-order polymorphic lambda calculus, is a lambda-calculus with polymorphic types of the form

$$\forall\alpha.t$$

where t is a type. The full syntax of terms and types in System F is given by the follow grammar:

$$\begin{aligned} \text{(types) } t &:= \alpha \mid t_1 \rightarrow t_2 \mid \forall\alpha.t \\ \text{(terms) } e &:= x \mid \lambda x : t.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e [t] \end{aligned}$$

The types consist of type variables (α), function types ($t_1 \rightarrow t_2$), and polymorphic types ($\forall\alpha.t$). The terms consist of term variables (x), term abstractions ($\lambda x : t.e$), term applications ($e_1 e_2$), type abstractions ($\Lambda\alpha.e$), and type applications ($e [t]$). We use Δ to stand for type contexts, which are finite sets of type variables, and Γ to stand for term contexts which are finite maps from term variables to types. The type formation judgment, $\Delta \vdash t : *$, holds whenever t is a type whose free variables are in Δ . The typing judgment, $\Delta; \Gamma \vdash e : t$ is determined by rules that correspond to natural deduction rules for second-order propositional logic, for which terms act as a proof language. The rules are:

$$\begin{aligned} &\frac{x : t \in \Gamma \quad \Delta \vdash t : *}{\Delta; \Gamma \vdash x : t} \text{ var} && \frac{\Delta; \Gamma, x : t_1 \vdash e : t_2 \quad \Delta \vdash t_1 : *}{\Delta; \Gamma \vdash \lambda x.e : t_1 \rightarrow t_2} \text{ abs} \\ &\frac{\Delta; \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Delta; \Gamma \vdash e_2 : t_1}{\Delta; \Gamma \vdash e_1 e_2 : t_2} \text{ app} && \frac{\Delta, \alpha : \sigma; \Gamma \vdash e : t}{\Delta; \Gamma \vdash e : \forall\alpha : \sigma.t} \text{ tabs} \\ &\frac{\Delta; \Gamma \vdash e : \forall\alpha : \sigma.t_2 \quad \Delta \vdash t_1 : \sigma}{\Delta; \Gamma \vdash e : [t_1/\alpha]t_2} \text{ tapp} \end{aligned}$$

Evaluation of programs in System F consists of reducing function application to the substitution of the function arguments for formal parameters in the body of the function. The rules for reduction are:

$$(\lambda x : t_1. e_1) e_2 \longrightarrow [e_2/x]e_1$$

$$(\Lambda \alpha. e) [t_1] \longrightarrow [t_1/\alpha]e$$

and these rules may be applied anywhere within a term. A term to which no reduction rules can be applied is said to be in normal form. A term is strongly normalizable if every reduction sequence starting from that term is finite. We will often treat derivations of judgments as mathematical objects. The notation $\mathcal{D} :: \mathcal{J}$ means that \mathcal{D} is a derivation of the judgment \mathcal{J} . The reduction rules of System F correspond to valid rules for transforming natural deduction proofs in second-order propositional logic:

$$\frac{\mathcal{D} :: \Delta; \Gamma, x : t_1 \vdash e_1 : t_2 \quad \Delta \vdash t_1 : *}{\Delta; \Gamma \vdash (\lambda x : t_1. e_1) : t_1 \rightarrow t_2} \text{abs} \quad \mathcal{E} :: \Delta; \Gamma \vdash e_2 : t_1}{\Delta; \Gamma \vdash (\lambda x : t_1. e_1) e_2 : t_2} \text{app} \Rightarrow [\mathcal{E}/x]\mathcal{D} :: \Delta; \Gamma \vdash [e_2/x]e_1 : t_2$$

$$\frac{\mathcal{D} :: \Delta, \alpha; \Gamma \vdash t_2}{\Delta; \Gamma \vdash (\Lambda \alpha. e) : \forall \alpha. t_2} \text{tabs} \quad \Delta \vdash t_1 : *}{\Delta; \Gamma \vdash (\Lambda \alpha. e) [t_1] : [t_1/\alpha]t_2} \text{tapp} \Rightarrow [t_1/\alpha]\mathcal{D} :: \Delta; \Gamma \vdash [t_1/\alpha]e : [t_1/\alpha]t_2$$

In System F, we can write functions which take arguments that are types. For example, we can write an identity function,

$$\text{id} = \Lambda \alpha. \lambda x : \alpha. x,$$

which can be given the type $\forall \alpha. \alpha \rightarrow \alpha$ and then can be used at any type, e.g.

$$\text{id [int] 1} = 1$$

$$\text{id [bool] true} = \text{true}$$

The universal polymorphism of System F is called *impredicative* because the quantifier ranges over all types, which leads to a sort of circularity. Terms which can be given the type $\forall \alpha. t$ can also be given the type $[t'/\alpha]t$ for any type t' , including $t' = \forall \alpha. t$. Therefore, any attempt to understand the meaning of $\forall \alpha. t$ by considering the meaning of $[t'/\alpha]t$ for each type t' is doomed to failure.

The standard way to show that all well-typed terms are strongly normalizing in simply-typed lambda calculus is to interpret types using reducibility predicates [14]. Each type is associated with a predicate that determines a set of strongly normalizing terms having that type and satisfying certain properties. One can then prove that all well-typed terms satisfy the predicate associated with their type, and are therefore strongly normalizing. The reducibility approach fails for System F because the circularity of its polymorphism does not allow for the reducibility predicate to be defined.

Girard proved strong normalization for System F using the method of reducibility candidates. Girard’s method works by defining a universe of predicates satisfying the important properties of the reducibility predicate. This universe of “reducibility candidates” is then used as the domain of quantification for interpreting polymorphic types. However, the proof requires elements of polymorphic types like $\forall\alpha.t$ act the same way at each $[t'/\alpha]t$, as opposed to acting differently depending on the type argument.

2.1 Type-Dependent Operations in System F

As Harper and Mitchell have shown in [7], extending System F with a conditional operator on types leads to a system which is not strongly normalizing. Suppose `typecond` is a constant with the typing rule

$$\frac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta; \Gamma \vdash e_2 : t_2}{\Delta; \Gamma \vdash \text{typecond } [t_1] [t_2] e_1 e_2 : t_2} \text{typecond}$$

and the reduction rules

$$\frac{}{\text{typecond } [t] [t] e_1 e_2 \longrightarrow e_1} \text{red-typecond-eq} \quad \frac{t, t' \text{ are distinct closed types}}{\text{typecond } [t] [t'] e_1 e_2 \longrightarrow e_2} \text{red-typecond-neq}$$

This seems like it could be a reasonable, and certainly useful, operator to add to System F, and the reduction rules do preserve the type of the term being reduced. However, this operator lets us define a term which reduces to itself, and is therefore not strongly normalizing. Consider the type $T = \forall\alpha.\alpha \rightarrow \alpha$ and the term

$$D = \Lambda\alpha.\lambda x : \alpha.\text{typecond } [T] [\alpha] (x [T] x) x$$

which can be given the type T . Then for the term $D [T] D$ we have

$$D [T] D \longrightarrow \text{typecond } [T] [T] (D [T] D) D \longrightarrow D [T] D$$

which shows that the system is not strongly normalizing. The fact that conditional branching on types leads to loss of strong normalization may lead one to conclude that intensional type analysis is not compatible with System F. In fact, calculi that extend System F with limited forms of intensional type analysis have been formulated and shown to be strongly normalizing. The systems of constraint types that we will study in this thesis can encode some intensional type analysis, via the GADT REP t described in the previous section, but that does not mean they cannot be strongly normalizing. In particular, it will not be possible to encode a polymorphic function with the type

$$\forall \alpha. \text{REP } \alpha$$

which would allow for the encoding of the `typecond` operator.

2.2 Impredicative Encodings of Algebraic Datatypes

The impredicative quantification of System F is extremely powerful. One can encode a number of useful data structures directly in System F using so-called *impredicative encodings*. The encodings for products and sums are extremely straightforward.

Example 2.1 (Products). The product type $t_1 \times t_2$ is a type of pairs or terms, (e_1, e_2) . Elements of the product type, $e : t_1 \times t_2$, can be decomposed using first project, `fst` e , and second projection, `snd` e . The typing rules associated with products are:

$$\frac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta; \Gamma \vdash e_2 : t_2}{\Delta; \Gamma \vdash (e_1, e_2) : t_1 \times t_2} \text{ pair}$$

$$\frac{\Delta; \Gamma \vdash e : t_1 \times t_2}{\Delta; \Gamma \vdash \text{fst } e : t_1} \text{ fst} \quad \frac{\Delta; \Gamma \vdash e : t_1 \times t_2}{\Delta; \Gamma \vdash \text{snd } e : t_2} \text{ snd}$$

The reduction rules associated with products are `fst` $(e_1, e_2) \longrightarrow e_1$ and `snd` $(e_1, e_2) \longrightarrow e_2$, along with standard congruence rules for `fst` e , `snd` e and (e_1, e_2) . This type and term constructors can be

encoded by:

$$\begin{aligned}
t_1 \times t_2 &:= \forall \alpha. (t_1 \rightarrow t_2 \rightarrow \alpha) \rightarrow \alpha \\
(e_1, e_2) &:= \Lambda \alpha. \lambda f : (t_1 \rightarrow t_2 \rightarrow \alpha). f \ e_1 \ e_2 \\
\text{fst } e &:= e [t_1] (\lambda x_1 : t_1. \lambda x_2 : t_2. x_1) \\
\text{snd } e &:= e [t_2] (\lambda x_1 : t_1. \lambda x_2 : t_2. x_2)
\end{aligned}$$

It is easy to see that the encodings are well-typed and that the reduction rules for $\text{fst } (e_1, e_2)$ and $\text{snd } (e_1, e_2)$ are simulated by multiple reduction steps using the encoding above.

Example 2.2 (Sums). The sum type $t_1 + t_2$ is either a left injection, $\text{inl}(e)$, or a right injection, $\text{inr}(e)$. Elements of the sum type, $e : t_1 + t_2$, can be decomposed using a case expression

$$\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$$

which branches depending on whether e evaluates to $\text{inl}(e')$ or $\text{inr}(e')$. The typing rules associated with sums are:

$$\begin{array}{c}
\frac{\Delta; \Gamma \vdash e : t_1}{\Delta; \Gamma \vdash \text{inl}(e) : t_1 + t_2} \text{ inl} \quad \frac{\Delta; \Gamma \vdash e : t_2}{\Delta; \Gamma \vdash \text{inr}(e) : t_1 + t_2} \text{ inr} \\
\frac{\Delta; \Gamma \vdash e : t_1 + t_2 \quad \Delta; \Gamma, x : t_1 \vdash e_1 : t \quad \Delta; \Gamma, y : t_2 \vdash e_2 : t}{\Delta; \Gamma \vdash \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} : t} \text{ case}
\end{array}$$

The reduction rules associated with sums are:

$$\begin{aligned}
\text{case inl}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} &\longrightarrow [e/x]e_1 \\
\text{case inr}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} &\longrightarrow [e/x]e_2
\end{aligned}$$

This type and term constructors can be encoded by:

$$\begin{aligned}
t_1 + t_2 &:= \forall \alpha. (t_1 \rightarrow \alpha) \rightarrow (t_2 \rightarrow \alpha) \rightarrow \alpha \\
\text{inl}(e) &:= \Lambda \alpha. \lambda f_1 : (t_1 \rightarrow \alpha). \lambda f_2 : (t_2 \rightarrow \alpha). f_1 \ e \\
\text{inr}(e) &:= \Lambda \alpha. \lambda f_1 : (t_1 \rightarrow \alpha). \lambda f_2 : (t_2 \rightarrow \alpha). f_2 \ e \\
\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} &:= e[t] (\lambda x : t_1. e_1) (\lambda y : t_2. e_2)
\end{aligned}$$

It is easy to see that the encodings are well-typed and that the reduction rules for

$$\text{case inl}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$$

and

$$\text{case inr}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$$

are simulated by multiple reduction steps using the encoding above.

In these encodings of products and sums, elements are defined as functions which mimic the elimination behavior of products and sums. Following a similar strategy, inductive datatypes like lists can be encoded.

Example 2.3 (Lists). The datatype of lists of type t has two constructors: a nullary constructor, nil , and a binary constructor, cons , which constructs a list given an element of type t and a list of type t . The destructor for lists, list-rec , takes a list, a term to replace nil with and a binary function to replace cons with. The associated typing rules are:

$$\frac{\Delta \vdash t : *}{\Delta; \Gamma \vdash \text{nil} : t \text{ list}} \text{ nil} \quad \frac{\Delta; \Gamma \vdash e_1 : t \quad \Delta; \Gamma \vdash e_2 : t \text{ list}}{\Delta; \Gamma \vdash \text{cons}(e_1, e_2) : t \text{ list}} \text{ cons}$$

$$\frac{\Delta; \Gamma \vdash e : t' \text{ list} \quad \Delta; \Gamma \vdash e_1 : t \quad \Delta; \Gamma, x_1 : t', x_2 : t \vdash e_2 : t}{\Delta; \Gamma \vdash \text{list-rec } e \text{ of } \{\text{nil} \rightarrow e_1; \text{cons}(x_1, x_2) \rightarrow e_2\} : t} \text{ list-rec}$$

The reduction rules associated with lists are:

$$\text{list-rec nil of } \{\text{nil} \rightarrow e_1; \text{cons}(x_1, x_2) \rightarrow e_2\} \longrightarrow e_1$$

$$\begin{aligned} & \text{list-rec cons}(e_1, e_2) \text{ of } \{\text{nil} \rightarrow e'_1; \text{cons}(x_1, x_2) \rightarrow e'_2\} \\ & \longrightarrow [e_1/x_1, \text{list-rec } e_2 \text{ of } \{\text{nil} \rightarrow e'_1; \text{cons}(x_1, x_2) \rightarrow e'_2\}/x_2]e'_2 \end{aligned}$$

This type and its term constructors can be encoded by:

$$\begin{aligned} t \text{ list} & := \forall \alpha. \alpha \rightarrow (t \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \\ \text{nil} & := \Lambda \alpha. \lambda x : \alpha. \lambda f : (t \rightarrow \alpha \rightarrow \alpha). x \\ \text{cons}(e_1, e_2) & := \Lambda \alpha. \lambda x : \alpha. \lambda f : (t \rightarrow \alpha \rightarrow \alpha). f e_1 (e_2 x f) \\ \text{list-rec } e \text{ of } \{\text{nil} \rightarrow e_1; \text{cons}(x_1, x_2) \rightarrow e_2\} & := e e_1 (\lambda x_1. \lambda x_2. e_2) \end{aligned}$$

It is easy to see that the encodings are well-typed and that the reduction rules for list-rec are simulated by multiple reduction steps using the encoding above.

3 System F with Constraint Types

In the rest of this work we will consider extensions of System F which include constraint types $(t_1 \dot{\leq} t_2)$ which can be assigned to terms that witness that t_1 is a subtype of t_2 . The systems we consider are Curry-style type-assignment systems, so there are no explicit type annotations in terms. However, when writing examples we will use explicit type annotations to make it clear that the terms are well-typed. Each annotated term can be unambiguously erased to a Curry-style term.

3.1 Definition of λ_{2C}

In this section we define λ_{2C} , a calculus for System F with constraint types.

Syntax

(sorts)	$\sigma := *$
(static terms)	$t := \alpha \mid t_1 \rightarrow t_2 \mid \forall \alpha : \sigma. t \mid (t_1 \dot{\leq} t_2)$
(static contexts)	$\Delta := \cdot \mid \Delta, \alpha : \sigma$
(terms)	$e := x \mid \lambda x. e \mid e_1 e_2 \mid \text{coerce}(e_1, e_2) \mid \text{refl} \mid \text{funI}(e_1, e_2) \mid \text{funEL}(e) \mid \text{funER}(e) \mid \text{allI}(e) \mid \text{allE}(e) \mid \text{subI}(e_1, e_2) \mid \text{subEL}(e) \mid \text{subER}(e) \mid \text{trans}(e_1, e_2)$
(dynamic contexts)	$\Gamma := \cdot \mid \Gamma, x : t$

The syntax of objects in λ_{2C} is broken into three levels: (dynamic) terms, static terms and sorts.

Terms include variables, x , drawn from an infinite set, lambda abstractions, $\lambda x. e$, and function applications, $e_1 e_2$. In addition to the standard λ -calculus terms, λ_{2C} also has proof terms which are witnesses for subtyping relationships between types. The proof terms are refl , $\text{funI}(e_1, e_2)$, $\text{funEL}(e)$, $\text{funER}(e)$, $\text{allI}(e)$, $\text{allE}(e)$, $\text{subI}(e_1, e_2)$, $\text{subEL}(e)$, $\text{subER}(e)$, $\text{trans}(e_1, e_2)$. Lastly, there is a coercion constructor $\text{coerce}(e_1, e_2)$ which uses a subtyping witness to change the type of a term.

The static terms include static variables, α , arrow types, $t_1 \rightarrow t_2$, universal types, $\forall \alpha : \sigma. t$, and constraint types, $(t_1 \dot{\leq} t_2)$. The only sort we will consider is $*$, the sort of types. Static and dynamic

contexts, Δ and Γ , are treated as finite maps. Extension of contexts, $\Delta, \alpha : \sigma$ and $\Gamma, x : t$, is only defined if the added variable (α and x respectively) is not already in the domain of the context.

The static variables α range over a countable set **StVars**, the term variables range over a countable set **TmVars**, the set of all terms is **Terms** the set of all sorts is **Sorts**, and the set of all static terms is **Statics**. We may abbreviate $\forall \alpha : *.t$ as $\forall \alpha.t$, particularly in examples.

Definition 3.1. Standard definitions

- Free variables of terms, static terms and each kind of context, $FV(e)$, $FV(t)$, $FV(\Gamma)$, $FV(\Delta)$
- Simultaneous capture-avoiding substitution for free variables in a term, $[e_1/x_1, \dots, e_n/x_n]e$, and a static term $[t_1/\alpha_1, \dots, t_n/\alpha_n]t$. For a substitution S , we use $S(e)$ or $S(t)$ to denote the result of applying of the substitution S to e and t respectively. We use $\text{dom}(S)$ to denote the set of variables that are substituted for by S .

Definition 3.2 (α -equivalence). \equiv_α is the least equivalence relation such that:

$$\begin{aligned} \lambda x.e &\equiv_\alpha \lambda y.[y/x]e && (\text{if } y \notin FV(e)) \\ [e_1/x]e &\equiv_\alpha [e_2/x]e && (\text{if } e_1 \equiv_\alpha e_2) \\ \\ \forall \alpha.t &\equiv_\alpha \forall \beta.[\beta/\alpha]t && (\text{if } \beta \notin FV(t)) \\ [t_1/\alpha]t &\equiv_\alpha [t_2/\alpha]t && (\text{if } t_1 \equiv_\alpha t_2) \end{aligned}$$

We will treat terms and static terms (types) as representatives of α -equivalence classes, and freely choose names for bound variables which do not clash with any other bound or free variables. When we write $e_1 = e_2$ or $t_1 = t_2$ we mean syntactic equality up to renaming of bound variables.

Semantics

The proof values of the language, a subtype of the terms, consist of constraint introduction forms applied to proof values.

Definition 3.3 (Proof Values).

$$(\text{proof values}) \quad p := \text{refl} \mid \text{funI}(p_1, p_2) \mid \text{allI}(p) \mid \text{subI}(p_1, p_2)$$

Definition 3.4 (Reduction). The rule **red-app** is the standard β -reduction rule. The rules **red-fun1**, **red-fun2**, **red-sub1**, **red-sub2**, and **red-all** reduce coercion proof redexes. The **red-coerce** rule throws away coercions by a value.

$$\begin{array}{c}
\frac{}{(\lambda x.e_2) e_1 \longrightarrow [e_1/x]e_2} \text{red-app} \quad \frac{}{\text{allE}(\text{allI}(e)) \longrightarrow e} \text{red-all} \\
\\
\frac{}{\text{funEL}(\text{funI}(e_1, e_2)) \longrightarrow e_1} \text{red-fun1} \quad \frac{}{\text{funER}(\text{funI}(e_1, e_2)) \longrightarrow e_2} \text{red-fun2} \\
\\
\frac{}{\text{subEL}(\text{subI}(e_1, e_2)) \longrightarrow e_1} \text{red-sub1} \quad \frac{}{\text{subER}(\text{subI}(e_1, e_2)) \longrightarrow e_2} \text{red-sub2} \\
\\
\frac{}{\text{coerce}(e, p) \longrightarrow e} \text{red-coerce}
\end{array}$$

Reduction is a congruence over all term constructors.

$$\begin{array}{c}
\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \text{red-appC1} \quad \frac{e_2 \longrightarrow e'_2}{e_1 e_2 \longrightarrow e_1 e'_2} \text{red-appC2} \quad \frac{e \longrightarrow e'}{\lambda x.e \longrightarrow \lambda x.e'} \text{red-lamC} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{funI}(e_1, e_2) \longrightarrow \text{funI}(e'_1, e_2)} \text{red-funIC1} \quad \frac{e_2 \longrightarrow e'_2}{\text{funI}(e_1, e_2) \longrightarrow \text{funI}(e_1, e'_2)} \text{red-funIC2} \\
\\
\frac{e \longrightarrow e'}{\text{funEL}(e) \longrightarrow \text{funEL}(e')} \text{red-funELC} \quad \frac{e \longrightarrow e'}{\text{funER}(e) \longrightarrow \text{funER}(e')} \text{red-funERC} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{subI}(e_1, e_2) \longrightarrow \text{subI}(e'_1, e_2)} \text{red-subIC1} \quad \frac{e_2 \longrightarrow e'_2}{\text{subI}(e_1, e_2) \longrightarrow \text{subI}(e_1, e'_2)} \text{red-subIC2} \\
\\
\frac{e \longrightarrow e'}{\text{subEL}(e) \longrightarrow \text{subEL}(e')} \text{red-subELC} \quad \frac{e \longrightarrow e'}{\text{subER}(e) \longrightarrow \text{subER}(e')} \text{red-subERC} \\
\\
\frac{e \longrightarrow e'}{\text{allI}(e) \longrightarrow \text{allI}(e')} \text{red-allIC} \quad \frac{e \longrightarrow e'}{\text{allE}(e) \longrightarrow \text{allE}(e')} \text{red-allEC} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{trans}(e_1, e_2) \longrightarrow \text{trans}(e'_1, e_2)} \text{red-transC1} \quad \frac{e_2 \longrightarrow e'_2}{\text{trans}(e_1, e_2) \longrightarrow \text{trans}(e_1, e'_2)} \text{red-transC2} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{coerce}(e_1, e_2) \longrightarrow \text{coerce}(e'_1, e_2)} \text{red-coerceC1} \quad \frac{e_2 \longrightarrow e'_2}{\text{coerce}(e_1, e_2) \longrightarrow \text{coerce}(e_1, e'_2)} \text{red-coerceC2}
\end{array}$$

In order to ensure that all closed, well-typed proof terms have a proof value as normal form we need to define reduction rules for the case of elimination terms applied to reflexivity as well as rules which implement transitivity elimination on proof terms.

$$\frac{}{\text{funEL}(\text{refl}) \longrightarrow \text{refl}} \text{red-funELrefl} \quad \frac{}{\text{funER}(\text{refl}) \longrightarrow \text{refl}} \text{ref-funERrefl}$$

$$\begin{array}{c}
\overline{\text{subEL}(\text{refl}) \longrightarrow \text{refl}} \quad \text{red-subELrefl} \quad \overline{\text{subER}(\text{refl}) \longrightarrow \text{refl}} \quad \text{red-subERrefl} \\
\overline{\text{allE}(\text{refl}) \longrightarrow \text{refl}} \quad \text{red-allErefl} \\
\overline{\text{trans}(\text{refl}, e) \longrightarrow e} \quad \text{red-trans-refl1} \quad \overline{\text{trans}(e, \text{refl}) \longrightarrow e} \quad \text{red-trans-refl2} \\
\overline{\text{trans}(\text{funI}(e_1, e_2), \text{funI}(e'_1, e'_2)) \longrightarrow \text{funI}(\text{trans}(e'_1, e_1), \text{trans}(e_2, e'_2))} \quad \text{red-trans-funI} \\
\overline{\text{trans}(\text{subI}(e_1, e_2), \text{subI}(e'_1, e'_2)) \longrightarrow \text{subI}(\text{trans}(e'_1, e_1), \text{trans}(e_2, e'_2))} \quad \text{red-trans-subI} \\
\overline{\text{trans}(\text{allI}(e), \text{allI}(e')) \longrightarrow \text{allI}(\text{trans}(e, e'))} \quad \text{red-trans-allI}
\end{array}$$

Definition 3.5 (Sorting).

$$\begin{array}{c}
\frac{(\alpha : \sigma) \in \Delta}{\Delta \vdash \alpha : \sigma} \quad \text{sort-var} \quad \frac{\Delta, \alpha : \sigma \vdash t : *}{\Delta \vdash \forall \alpha : \sigma. t : *} \quad \text{sort-all} \\
\frac{\Delta \vdash t_1 : * \quad \Delta \vdash t_2 : *}{\Delta \vdash t_1 \rightarrow t_2 : *} \quad \text{sort-arrow} \quad \frac{\Delta \vdash t_1 : * \quad \Delta \vdash t_2 : *}{\Delta \vdash (t_1 \dot{\leq} t_2) : *} \quad \text{sort-sub}
\end{array}$$

Definition 3.6 (Type Assignment). The typing judgment has the form $\Delta; \Gamma \vdash e : t$ where the rules are designed to ensure that $\Delta \vdash t : *$.

$$\begin{array}{c}
\frac{x : t \in \Gamma \quad \Delta \vdash t : *}{\Delta; \Gamma \vdash x : t} \quad \text{var} \quad \frac{\Delta; \Gamma, x : t_1 \vdash e : t_2 \quad \Delta \vdash t_1 : *}{\Delta; \Gamma \vdash \lambda x. e : t_1 \rightarrow t_2} \quad \text{abs} \\
\frac{\Delta; \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Delta; \Gamma \vdash e_2 : t_1}{\Delta; \Gamma \vdash e_1 e_2 : t_2} \quad \text{app} \quad \frac{\Delta, \alpha : \sigma; \Gamma \vdash e : t}{\Delta; \Gamma \vdash e : \forall \alpha : \sigma. t} \quad \text{tabs} \\
\frac{\Delta; \Gamma \vdash e : \forall \alpha : \sigma. t_2 \quad \Delta \vdash t_1 : \sigma}{\Delta; \Gamma \vdash e : [t_1/\alpha]t_2} \quad \text{tapp} \quad \frac{\Delta; \Gamma \vdash e_1 : t \quad \Delta; \Gamma \vdash e_2 : (t \dot{\leq} t')}{\Delta; \Gamma \vdash \text{coerce}(e_1, e_2) : t'} \quad \text{coerce} \\
\frac{\Delta \vdash t : *}{\Delta; \Gamma \vdash \text{refl} : (t \dot{\leq} t)} \quad \text{refl} \quad \frac{\Delta; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1) \quad \Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta; \Gamma \vdash \text{funI}(e_1, e_2) : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)} \quad \text{funI} \\
\frac{\Delta; \Gamma \vdash e : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)}{\Delta; \Gamma \vdash \text{funEL}(e) : (t'_1 \dot{\leq} t_1)} \quad \text{funEL} \quad \frac{\Delta; \Gamma \vdash e : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)}{\Delta; \Gamma \vdash \text{funER}(e) : (t_2 \dot{\leq} t'_2)} \quad \text{funER} \\
\frac{\Delta; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1) \quad \Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta; \Gamma \vdash \text{subI}(e_1, e_2) : ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))} \quad \text{subI} \quad \frac{\Delta; \Gamma \vdash e : ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))}{\Delta; \Gamma \vdash \text{subEL}(e) : (t'_1 \dot{\leq} t_1)} \quad \text{subEL} \\
\frac{\Delta; \Gamma \vdash e : ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))}{\Delta; \Gamma \vdash \text{subER}(e) : (t_2 \dot{\leq} t'_2)} \quad \text{subER} \quad \frac{\Delta, \alpha : \sigma; \Gamma \vdash e : (t \dot{\leq} t')}{\Delta; \Gamma \vdash \text{allI}(e) : (\forall \alpha : *. t \dot{\leq} \forall \alpha : \sigma. t')} \quad \text{allI}
\end{array}$$

$$\frac{\Delta, \Gamma \vdash e : (\forall \alpha : \sigma. t \dot{\leq} \forall \alpha : \sigma. t') \quad \Delta \vdash s : \sigma}{\Delta; \Gamma \vdash \text{allE}(e) : ([s/\alpha]t \dot{\leq} [s/\alpha]t')} \text{allE}$$

$$\frac{\Delta; \Gamma \vdash e_1 : (t \dot{\leq} t') \quad \Delta; \Gamma \vdash e_2 : (t' \dot{\leq} t'')}{\Delta; \Gamma \vdash \text{trans}(e_1, e_2) : (t \dot{\leq} t'')} \text{trans}$$

Where `tabs` and `alll` have the side-condition that $\alpha \notin FV(\Gamma)$.

The rules `var`, `abs`, `app`, `tabs`, and `tapp` are the standard typing rules of System F. The main novelty of the type system is the types of the form $(t_1 \dot{\leq} t_2)$ which are types for proofs that t_1 is a subtype of t_2 . The `coerce` rule allows us to change the type of a term by making use of a proof of subtyping. The remaining typing rules allow us to check proofs of type equality. Proofs of subtyping between function and subtyping types have identical forms, they are covariant in the right-hand side and contra-variant in the left hand-side. Subtyping types between quantified types are introduced and eliminated in a fashion that is parallel to the introduction and elimination of quantified types themselves.

3.2 Impredicative Encodings of GADTs

In this section we describe how constraint types may be used to encode GADTs. To encode GADTs, we follow the same strategy used for encoding regular algebraic datatypes, and we simply consider each constructor of the datatypes to take additional arguments of constraint type. We make use of $t_1 \dot{=} t_2$ in place of the combination of $(t_1 \dot{\leq} t_2)$ and $(t_2 \dot{\leq} t_1)$, and use `refl` as a proof of $t_1 \dot{=} t_2$. This is done purely for brevity and can be expanded to use subtyping proofs.

Example 3.7 (TERM). The TERM example from the first section can be encoded as:

$$\begin{aligned} \text{TERM}(t) \quad &:= \quad \forall \alpha : *. ((t \dot{=} \text{int}) \rightarrow \text{int} \rightarrow \alpha) \rightarrow \\ &\quad ((t \dot{=} \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \\ &\quad ((t \dot{=} \text{bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \\ &\quad ((t \dot{=} \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \end{aligned}$$

Each encoding is for a data element of type $\text{REP}(t)$.

$$\text{TERMint}(n) := \Lambda \alpha : *.$$

$$\lambda f_1 : ((t \doteq \text{int}) \rightarrow \text{int} \rightarrow \alpha)$$

$$\lambda f_2 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_3 : ((t \doteq \text{bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_4 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$f_1 \text{ refl } n$$

$$\text{TERMplus}(tm_1, tm_2) := \Lambda \alpha : *.$$

$$\lambda f_1 : ((t \doteq \text{int}) \rightarrow \text{int} \rightarrow \alpha)$$

$$\lambda f_2 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_3 : ((t \doteq \text{bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_4 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$f_2 \text{ refl } (tm_1 f_1 f_2 f_3 f_4) (tm_2 f_1 f_2 f_3 f_4)$$

$$\text{TERMleq}(tm_1, tm_2) := \Lambda \alpha : *.$$

$$\lambda f_1 : ((t \doteq \text{int}) \rightarrow \text{int} \rightarrow \alpha)$$

$$\lambda f_2 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_3 : ((t \doteq \text{bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$\lambda f_4 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

$$f_3 \text{ refl } (tm_1 f_1 f_2 f_3 f_4) (tm_2 f_1 f_2 f_3 f_4)$$

$\text{TERMite}(tm_1, tm_2, tm_3) := \Lambda\alpha : *.$

$\lambda f_1 : ((t \doteq \text{int}) \rightarrow \text{int} \rightarrow \alpha)$

$\lambda f_2 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$

$\lambda f_3 : ((t \doteq \text{bool}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$

$\lambda f_4 : ((t \doteq \text{int}) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$

$f_4 \text{ refl } (tm_1 f_1 f_2 f_3 f_4) (tm_2 f_1 f_2 f_3 f_4) (tm_3 f_1 f_2 f_3 f_4)$

Example 3.8 (REP). The REP example from the first section can be encoded as follows:

$\text{REP}(t) := \forall\alpha : *. ((t \doteq ()) \rightarrow \alpha) \rightarrow$

$(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow$

$(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow$

$(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$

Each encoding is for a data element of type $\text{REP}(t)$.

$\text{REPunit} := \Lambda\alpha : *.$

$\lambda f_1 : ((t \doteq ()) \rightarrow \alpha)$

$\lambda f_2 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$

$\lambda f_3 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$

$\lambda f_4 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha).$

$f_1 \text{ refl}$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\begin{aligned}
\text{REPfun}(r_1, r_2) &:= \Lambda \alpha : *. \\
&\lambda f_1 : ((t \doteq ()) \rightarrow \alpha) \\
&\lambda f_2 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_3 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_4 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha). \\
&f_2 [t_1] [t_2] \text{refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4)
\end{aligned}$$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\begin{aligned}
\text{REPprod}(r_1, r_2) &:= \Lambda \alpha : *. \\
&\lambda f_1 : ((t \doteq ()) \rightarrow \alpha) \\
&\lambda f_2 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_3 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_4 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha). \\
&f_3 [t_1] [t_2] \text{refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4)
\end{aligned}$$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\begin{aligned}
\text{REPSum}(r_1, r_2) &:= \Lambda \alpha : *. \\
&\lambda f_1 : ((t \doteq ()) \rightarrow \alpha) \\
&\lambda f_2 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_3 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \\
&\lambda f_4 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha). \\
&f_4 [t_1] [t_2] \text{refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4)
\end{aligned}$$

Unfortunately, these encodings are not particularly useful, because they only allow for defining functions over GADTs that have a fixed return type. One cannot encode, for example, an evaluation

function of type $\forall\alpha.\text{TERM}(\alpha) \rightarrow \alpha$. The reason for this is that TERM is a heterogeneous datatype, so when evaluating an element of type $\text{TERM}(\text{int})$, we may also have to evaluate an element of type $\text{TERM}(\text{bool})$ recursively, which we cannot achieve with this encoding. In order to remedy this limitation, one could extend the language with type-level functions and higher-order polymorphism. Assuming such an extension, we can encode GADTs with useful elimination behavior.

Example 3.9 (REP). The REP example from the first section can be encoded as follows:

$$\begin{aligned} \text{REP}(t) \quad &:= \quad \forall\alpha : * \rightarrow *. ((t \doteq ()) \rightarrow (\alpha t)) \rightarrow \\ &(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \rightarrow \\ &(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \rightarrow \\ &(\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \rightarrow (\alpha t) \end{aligned}$$

Each encoding is for a data element of type $\text{REP}(t)$.

$$\begin{aligned} \text{REPunit} \quad &:= \quad \Lambda\alpha : * \rightarrow *. \\ &\lambda f_1 : ((t \doteq ()) \rightarrow (\alpha t)) \\ &\lambda f_2 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &\lambda f_3 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &\lambda f_4 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &f_1 \text{ refl} \end{aligned}$$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\begin{aligned} \text{REPfun}(r_1, r_2) \quad &:= \quad \Lambda\alpha : * \rightarrow *. \\ &\lambda f_1 : ((t \doteq ()) \rightarrow (\alpha t)) \\ &\lambda f_2 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &\lambda f_3 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &\lambda f_4 : (\forall\alpha_1 : *. \forall\alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t)) \\ &f_2 [t_1] [t_2] \text{ refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4) \end{aligned}$$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\text{REPprod}(r_1, r_2) := \Lambda \alpha : * \rightarrow *$$

$$\lambda f_1 : ((t \doteq ()) \rightarrow (\alpha t))$$

$$\lambda f_2 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$\lambda f_3 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$\lambda f_4 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$f_3 [t_1] [t_2] \text{ refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4)$$

For $r_1 : \text{REP}(t_1)$ and $r_2 : \text{REP}(t_2)$:

$$\text{REPsum}(r_1, r_2) := \Lambda \alpha : * \rightarrow *$$

$$\lambda f_1 : ((t \doteq ()) \rightarrow (\alpha t))$$

$$\lambda f_2 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \rightarrow \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$\lambda f_3 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 \times \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$\lambda f_4 : (\forall \alpha_1 : *. \forall \alpha_2 : *. (t \doteq \alpha_1 + \alpha_2) \rightarrow (\alpha \alpha_1) \rightarrow (\alpha \alpha_2) \rightarrow (\alpha t))$$

$$f_4 [t_1] [t_2] \text{ refl } (r_1 f_1 f_2 f_3 f_4) (r_2 f_1 f_2 f_3 f_4)$$

If we add to the language string constants, “string”, and concatenation $s_1 + s_2$, then we can program

a generic function which creates a string for a term by:

$$\begin{aligned}
\text{to_string} &:= \Lambda \alpha : *. \lambda r : \text{REP}(\alpha). \\
& r [\lambda \beta : *. \beta \rightarrow \text{string}] (\lambda x : (t \dot{=} ())). \lambda y : t. \text{“unit”} \\
& (\Lambda \alpha_1 : *. \Lambda \alpha_2 : *. \lambda x : (t \dot{=} \alpha_1 \rightarrow \alpha_2). \lambda_ : (\alpha_1 \rightarrow \text{string}). \lambda_ : (\alpha_2 \rightarrow \text{string}). \\
& \quad \lambda y : t. \text{“fun”}) \\
& (\Lambda \alpha_1 : *. \Lambda \alpha_2 : *. \lambda x : (t \dot{=} \alpha_1 \times \alpha_2). \lambda f_1 : (\alpha_1 \rightarrow \text{string}). \lambda f_2 : (\alpha_2 \rightarrow \text{string}). \\
& \quad \lambda y : t. \text{“} + f_1(\text{fst}(\text{coerce}(y, x))) + \text{“,”} + f_2(\text{snd}(\text{coerce}(y, x))) + \text{“”}) \\
& (\Lambda \alpha_1 : *. \Lambda \alpha_2 : *. \lambda x : (t \dot{=} \alpha_1 + \alpha_2). \lambda f_1 : (\alpha_1 \rightarrow \text{string}). \lambda f_2 : (\alpha_2 \rightarrow \text{string}). \\
& \quad \lambda y : t. \text{case } \text{coerce}(y, x) \text{ of} \\
& \quad \quad \{\text{inl}(x_1) \rightarrow \text{“inl”} + f_1(x_1) + \text{“”}, \\
& \quad \quad \text{inr}(x_2) \rightarrow \text{“inr”} + f_2(x_2) + \text{“”}\})
\end{aligned}$$

In this work we do not fully investigate the extension to System F_ω with constraint types. However, we do believe that such an extension is possible, and can be shown to be strongly normalizing without much complication. We will leave such an extension as future work.

3.3 Type Soundness and Basic Meta-theory of λ_{2C}

In this section we will prove some basic properties of λ_{2C} including type soundness. Several of the lemmas we prove in this section will be used in proving strong normalization.

It is useful to have a set of rules for assigning constraint types to proof values in which type abstraction and application cannot be used. This way we have a syntax-directed characterization of subtyping proofs, so inversion is free. Note that proof values do not contain free variables, so we do not need a term context for this judgment.

Definition 3.10 (Subtyping Value Derivations).

$$\begin{array}{c}
\frac{\Delta \vdash t : *}{\Delta \vdash \text{refl} : (t \dot{\leq} t)} \text{subl-refl} \qquad \frac{\Delta \vdash e_1 : (t'_1 \dot{\leq} t_1) \quad \Delta \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta \vdash \text{funI}(e_1, e_2) : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)} \text{subt-funI} \\
\\
\frac{\Delta \vdash e_1 : (t'_1 \dot{\leq} t_1) \quad \Delta \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta \vdash \text{subI}(e_1, e_2) : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)} \text{subt-subI} \qquad \frac{\Delta, \alpha : \sigma \vdash e : (t \dot{\leq} t')}{\Delta \vdash \text{allI}(e) : (\forall \alpha : *. t \dot{\leq} \forall \alpha : \sigma. t')} \text{subt-allI}
\end{array}$$

The following α -renaming lemma serves as a sanity check for our convention of identifying terms and types which differ only in the names of bound variables.

Lemma 3.11 (α -renaming). Each of the judgments admits α -renaming in the contexts. Given an infinite computable set \mathcal{V}_x of term variables and \mathcal{V}_α of type variables.

1. If $\mathcal{D} :: \Delta, \alpha_1 : \sigma_1 \vdash t : \sigma$ then for all but finitely many $\alpha \in \mathcal{V}_\alpha$, $\Delta, \alpha : \sigma_1 \vdash [\alpha/\alpha_1]t : \sigma$.
2. If $\mathcal{D} :: \Delta, \alpha_1 : \sigma_1; \Gamma \vdash e : t$ and $\alpha_1 \notin FV(\Gamma)$ then for all but finitely many $\alpha \in \mathcal{V}_\alpha$, $\Delta, \alpha : \sigma_1; \Gamma \vdash e : [\alpha/\alpha_1]t$.
3. If $\mathcal{D} :: \Delta, \alpha_1 : \sigma_1 \vdash p : t$ then for all but finitely many $\alpha \in \mathcal{V}_\alpha$, $\Delta, \alpha : \sigma_1 \vdash p : [\alpha/\alpha_1]t$.
4. If $\mathcal{D} :: \Delta; \Gamma, x_1 : t_1 \vdash e : t$ then for all but finitely many $x \in \mathcal{V}_x$, $\Delta; \Gamma, x : t_1 \vdash [x/x_1]e : t$.

Proof. Each part is proved by induction on the structure of \mathcal{D} , with part 2 relying on part 1. \square

Lemma 3.12 (Weakening). Each of the judgments admits weakening:

1. If $\mathcal{D} :: \Delta \vdash t : \sigma$ then $\Delta, \alpha : \sigma_1 \vdash t : \sigma$.
2. If $\mathcal{D} :: \Delta; \Gamma \vdash e : t$ and $\Delta \vdash t_1 : *$ then. $\Delta; \Gamma, x_1 : t_1 \vdash e : t$.
3. If $\mathcal{D} :: \Delta; \Gamma \vdash e : t$ then $\Delta, \alpha : \sigma; \Gamma \vdash e : t$.

Proof. Parts 1 and 2 can be proved independently by induction on the structure of \mathcal{D} . Part 3 is by induction on the structure of \mathcal{D} , making use of part 1. \square

Lemma 3.13 (Substitution on Type Assignment). If

$$\mathcal{D} :: \Delta; \Gamma, x_1 : t_1 \vdash e : t$$

and

$$\mathcal{E} :: \Delta; \Gamma \vdash e_1 : t_1$$

then

$$\Delta; \Gamma \vdash [e_1/x]e : t$$

Proof. By induction on the structure of the derivation \mathcal{D} . In the **var** case, we use \mathcal{E} , in the **abs** case we use weakening in the term context (Lemma 3.12) of \mathcal{E} and use the fact that $\Gamma, x_1 : t_1, x_2 : t_2 =$

$\Gamma, x_2 : t_2, x_1 : t_1$. For the **tabs** case we use weakening in the type context of \mathcal{E} . In each other case the conclusion follows from the induction hypothesis by applying the inference rule for that case. We make use of the fact that $\Gamma, x_1 : t_1, x_2 : t_2 = \Gamma, x_2 : t_2, x_1 : t_1$ in the **abs** case. \square

In order to prove that we can substitute types for type variables in a derivation, we will first need two lemmas.

Lemma 3.14 (Substitution on Sorting). If $\mathcal{D} :: \Delta, \alpha_1 : \sigma_1 \vdash t : \sigma$ and $\Delta \vdash t_1 : \sigma_1$ then $\Delta \vdash [t_1/\alpha_1]t : \sigma$.

Proof. By Induction on the structure of \mathcal{D} . \square

Lemma 3.15 (Substitution Commutes). If $\beta \notin \text{dom}(T)$ and $\beta \notin FV(T)$ then

$$T([t_1/\beta](t)) = [T(t_1)/\beta](T(t))$$

Proof. By induction on the structure of t .

case: $t = \beta$. Because β does not get substituted for by T :

$$([T(t_1)/\beta](T(\beta))) = [T(t_1)/\beta]\beta = T(t_1) = T([t_1/\beta](\beta))$$

case: $t = \alpha_i \in \text{dom}(T)$. Because $\beta \notin FV(T)$:

$$[T(t_1)/\beta](T(\alpha_i)) = T(\alpha_i)$$

and because $\beta \neq \alpha_i$

$$T(\alpha_i) = T([t_1/\beta]\alpha_i)$$

case: $t = \alpha \notin \text{dom}(T), \alpha \neq \beta$.

$$T([t_1/\beta](\alpha)) = \alpha = [T(t_1)/\beta](T(\alpha))$$

Each other case follows from the IH, using the definition of substitution. \square

Lemma 3.16 (Assigned Types are Well-Sorted). If $\mathcal{D} :: \Delta; \Gamma \vdash e : t$ then $\Delta \vdash t : *$.

Proof. By induction on the structure of \mathcal{D} . **tapp** case requires the use of Lemma 3.14, the other cases are straightforward. \square

Lemma 3.17 (Substitution for Type Variables in Type Assignments). If $\mathcal{D} :: \Delta, \alpha : \sigma; \Gamma \vdash e : t$ and $\Delta \vdash t_1 : \sigma$ then $\Delta; [t_1/\alpha]\Gamma \vdash e : [t_1/\alpha]t$.

Proof. By induction on the structure of \mathcal{D} . The only non-trivial case is:

$$\frac{\Delta, \alpha : \sigma; \Gamma \vdash e : \forall \beta : \sigma_1. t \quad \Delta, \alpha : \sigma \vdash t' : *}{\Delta, \alpha : \sigma; \Gamma \vdash e : [t'/\beta]t} \text{tapp}$$

We can assume β is fresh. From the IH, we have $\Delta; [t_1/\alpha]\Gamma \vdash e : [t_1/\alpha]\forall \beta : \sigma_1. t$, which is equal to $\mathcal{D}' :: \Delta; [t_1/\alpha]\Gamma \vdash e : \forall \beta : \sigma_1. [t_1/\alpha]t$. By Lemma 3.14 we have $\mathcal{E}' :: \Delta \vdash [t_1/\alpha]t' : *$. We apply **tapp** to \mathcal{D}' and \mathcal{E} to get

$$\Delta; [t_1/\alpha]\Gamma \vdash e : [[t_1/\alpha]t'/\beta][t_1/\alpha]t$$

By Lemma 3.15

$$[[t_1/\alpha]t'/\beta][t_1/\alpha]t = [t_1/\alpha]([t'/\beta]t)$$

which concludes the case. \square

Lemma 3.18. If $\mathcal{D} :: \Delta; \Gamma \vdash p : t$ and for some t'_1, t'_2 , $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (t'_1 \dot{\leq} t'_2)$ (for $n \geq 0$). Then $\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n \vdash p : (t'_1 \dot{\leq} t'_2)$

Proof. By induction on the structure of \mathcal{D} . For each form of p there are three cases to consider for the last rule: the introduction rule corresponding to that case, **tabs**, and **tapp**.

case: refl. Then $t'_1 = t'_2$ and $\Delta \vdash \text{refl} : (t'_1 \dot{\leq} t'_2)$

case: funl, subl or alll. In each case we have $n = 0$ and the conclusion follows from applying the IH to the premises of the rule.

case: tabs.

$$\frac{\Delta, \alpha : \sigma; \Gamma \vdash e : t_1}{\Delta; \Gamma \vdash e : \forall \alpha : \sigma. t_1} \text{tabs}$$

Then it must be that $n > 0$, and t_1 has the correct form to apply the IH, from which the conclusion follows directly.

case: tapp. Then it must be that $n \geq 0$.

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha : \sigma. t_2 \quad \Delta \vdash t_1 : \sigma}{\Delta; \Gamma \vdash e : [t_1/\alpha]t_2} \text{ tapp}$$

Since $[t_1/\alpha]t_2 = t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (t'_1 \dot{\leq} t'_2)$, $\forall \alpha : \sigma. t_2$ has the correct form to apply the IH, and the conclusion follows from Lemma 3.17. \square

The opposite direction also holds.

Lemma 3.19. If $\Delta \vdash p : (t_1 \dot{\leq} t_2)$ then $\Delta; \cdot \vdash p : (t_1 \dot{\leq} t_2)$

Proof. Each rule in the judgment $\Delta \vdash p : (t_1 \dot{\leq} t_2)$ has the same form as a typing rule which leaves the term context unchanged. \square

Lemma 3.20 (Substitution for Type Variables in Proof Value Judgement). If $\mathcal{D} :: \Delta, \alpha : \sigma \vdash e : t$ and $\Delta \vdash t_1 : \sigma$ then $\Delta \vdash e : [t_1/\alpha]t$.

Proof. Follows from Lemmas 3.19, 3.17, and 3.18. \square

Given the way we have formulated the rules for subtyping constraints, closed subtyping proofs only exist for constraints where the types that are actually equal.

Lemma 3.21 (Subtyping is Equality). If $\mathcal{D} :: \Delta \vdash p : (t_1 \dot{\leq} t_2)$ then $t_1 = t_2$.

Proof. By induction on the structure of \mathcal{D} . \square

Type Soundness

We will prove type soundness via the usual ‘‘preservation’’ and ‘‘progress’’ lemmas. When proving progress, we will use a generalized definition for values which includes any term whose top-level constructor is lambda. The progress lemma will prove that any term that is well-typed and not a generalized value must have a reduction. Note that terms which are generalized values may have applicable reductions under lambda.

Lemma 3.22 (Subtyping Soundness). If $\Delta; \Gamma \vdash e : t_1$ and $\Delta' \vdash p : (t_1 \dot{\leq} t_2)$ then $\Delta; \Gamma \vdash e : t_2$.

Proof. By Lemma 3.21 $t_1 = t_2$. \square

Lemma 3.23 (Inversion on Typing). Suppose $\mathcal{D} :: \Delta; \Gamma \vdash e : t$, then

1. if $e = \lambda x.e'$ then $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. t_1 \rightarrow t_2$ and

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n; \Gamma, x : t_1 \vdash e' : t_2$$

2. if $e = \text{refl}$ then $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (t' \dot{\leq} t')$

3. if $e = \text{funI}(e_1, e_2)$ then $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)$ and

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1)$$

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)$$

4. if $e = \text{subI}(e_1, e_2)$ then $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))$ and

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1)$$

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)$$

5. if $e = \text{allI}(e')$ then $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (\forall \alpha : \sigma. t_1 \dot{\leq} \forall \alpha : \sigma. t'_1)$ and

$$\Delta, \alpha_1 : \sigma_1, \dots, \alpha_n : \sigma_n, \alpha : \sigma; \Gamma \vdash e' : (t_1 \dot{\leq} t'_1)$$

Proof. By induction on the structure of \mathcal{D} . In each cases, the only rules that can apply are **tabs**, **tapp** or the introduction rule corresponding to the top-level term constructor of e . The **tabs** case follows directly from the IH. For the **tapp** case we use the IH and Lemma 3.17. The cases for rules corresponding to top-level term constructors follow easily from the IH. \square

Lemma 3.24 (Preservation). If $\mathcal{D} :: \Delta; \Gamma \vdash e : t$ and $e \longrightarrow e'$ then $\Delta; \Gamma \vdash e' : t$

Proof. By induction on the structure of \mathcal{D} making use of inversion on the derivation of reduction.

case: var. No reduction rules apply, so this case is impossible.

case: tabs, tapp. Follows easily from the IH.

case: abs. Then $e = \lambda x.e_1$, $t = t_1 \rightarrow t_2$, $\Delta; \Gamma, x : t_1 \vdash e_1 : t_2$. The only reduction rule which applies is **red-lamC**, so we have $e_1 \longrightarrow e'_1$. By the IH we have $\Delta; \Gamma, x : t_1 \vdash e'_1 : t_2$, so by **abs** we have

$\Delta; \Gamma \vdash \lambda x.e'_1 : t_1 \rightarrow t_2$.

case: app. Then $e = e_1 e_2$ and for some t' , $\Delta; \Gamma \vdash e_1 : t' \rightarrow t$ and $\Delta; \Gamma \vdash e_2 : t'$. If the reduction was a congruence rule, then the IH gives us the desired conclusion. Otherwise the reduction rule was **red-app**, so $e_1 = \lambda x.e'_1$. By Lemma 3.23, we have $\Delta; \Gamma, x : t' \vdash e'_1 : t$. By Lemma 3.13 we have $\Delta; \Gamma \vdash [e_2/x]e'_1 : t$.

case: coerce. Then $e = \text{coerce}(e_1, e_2)$, and for some t_1 , $\Delta; \Gamma \vdash e_1 : t_1$ and $\Delta; \Gamma \vdash e_2 : (t_1 \dot{\leq} t)$. If the reduction is a congruence rule, then the IH gives the desired conclusion. Otherwise the reduction must be from rule **red-coerce**, so e_2 is a proof value. By Lemma 3.18 there exists $\Delta' \supseteq \Delta$ such that $\Delta' \vdash e_2 : (t_1 \dot{\leq} t)$, so by Lemma 3.22 we have $\Delta; \Gamma \vdash e_1 : t$.

case: trans. Then $e = \text{trans}(e_1, e_2)$, and for some $t_1, t_2, t_3, t = (t_1 \dot{\leq} t_3)$, $\Delta; \Gamma \vdash e_1 : (t_1 \dot{\leq} t_2)$ and $\Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t_3)$. If the reduction is a congruence rule, then the IH gives the desired conclusion. Otherwise the reduction is one of **red-trans-refl1**, **red-trans-refl2**, **red-trans-funI**, **red-trans-subI** or **red-trans-allI**. Suppose it is **red-trans-allI**. Then $e_1 = \text{allI}(e'_1)$, $e_2 = \text{allI}(e'_2)$ and by Lemma 3.23 we have $t_1 = \forall \alpha : \sigma.t'_1$, $t_2 = \forall \alpha : \sigma.t'_2$, $t_3 = \forall \alpha : \sigma.t'_3$, $\Delta, \alpha : \sigma \vdash e'_1 : (t'_1 \dot{\leq} t'_2)$ and $\Delta, \alpha : \sigma \vdash e'_2 : (t'_2 \dot{\leq} t'_3)$. So by **trans**, $\Delta, \alpha : \sigma \vdash \text{trans}(e'_1, e'_2) : (t'_1 \dot{\leq} t'_3)$ and by **allI**, $\Delta, \alpha : \sigma \vdash \text{allI}(\text{trans}(e'_1, e'_2)) : (t_1 \dot{\leq} t_3)$. The proofs for the other reduction rules are similar.

For the rest of the cases, reductions from congruence rules follow directly from the IH, and other reductions follow easily from Lemma 3.23. □

Definition 3.25 (Values). The set of (generalized) values consists of introduction forms applied to values and terms whose top-level constructor is lambda.

$$(\text{values}) \ v \quad := \quad \lambda x.e \mid \text{refl} \mid \text{funI}(v_1, v_2) \mid \text{subI}(v_1, v_2) \mid \text{allI}(v)$$

Lemma 3.26 (Canonical Forms). Suppose $\mathcal{D} :: \Delta; \cdot \vdash v : t$, then

1. if $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. t_1 \rightarrow t_2$ then $v = \lambda x.e$ for some e ,
2. if $t = \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. (t_1 \dot{\leq} t_2)$ then either $v = \text{refl}$, or
 - $t_1 = t_{11} \rightarrow t_{12}$ and $t_2 = t_{21} \rightarrow t_{22}$ and $v = \text{funI}(v_1, v_2)$, or
 - $t_1 = (t_{11} \dot{\leq} t_{12})$ and $t_2 = (t_{21} \dot{\leq} t_{22})$ then $v = \text{subI}(v_1, v_2)$, or

- $t_1 = \forall\alpha : \sigma.t'_1$ and $t_2 = \forall\alpha : \sigma.t'_2$ and $v = \text{allI}(v')$.

Proof. By a straightforward induction on the structure of \mathcal{D} . □

Lemma 3.27 (Progress). If $\mathcal{D} :: \Delta; \cdot \vdash e : t$ then either e is a value or for some e' , $e \longrightarrow e'$.

Proof. By induction on the structure of \mathcal{D} .

case: var. The term context is empty, so this rule cannot apply.

case: tabs, tapp. The conclusion comes directly from the IH.

case: abs. Then $e = \lambda x.e'$, which is a value.

case: app. Then $e = e_1 e_2$. By the IH, either e_1 and e_2 are values or else at least one can be reduced. If either e_1 or e_2 can be reduced, then e can be reduced by applying to corresponding congruence rule. If both are values, then by Lemma 3.26, $e_1 = \lambda x.e'_1$ for some e'_1 so the reduction rule **red-app** applies.

case: coerce. Then $e = \text{coerce}(e_1, e_2)$. By the IH, either e_1 and e_2 are values or else at least one can be reduced. If either e_1 or e_2 can be reduced, then e can be reduced by applying to corresponding congruence rule. If both are values then **red-coerce** must apply.

case: refl. Then $e = \text{refl}$, which is a value.

case: funI. Then $e = \text{funI}(e_1, e_2)$. By the IH, either e_1 and e_2 are values or else at least one can be reduced. If either e_1 or e_2 can be reduced, then e can be reduced by applying to corresponding congruence rule. If both are values then $\text{funI}(e_1, e_2)$ is a value.

case: allI, subI. Similar to the previous case.

case: funEL. Then $e = \text{funEL}(e')$. By the IH, either e' is a value or it can be reduced. If e' can be reduced then e can be reduced using a congruence rule. If e' is a value then by Lemma 3.26 it must either be **refl** or of the form $\text{funI}(e_1, e_2)$, and in either case a reduction rule applies.

case: funER, subEL, subER, allE. These cases are similar to the previous case.

case: trans. Then $e = \text{trans}(e_1, e_2)$ and for some t_1, t_2, t_3 , $\Delta; \cdot \vdash e_1 : (t_1 \dot{\leq} t_2)$, and $\Delta; \cdot \vdash e_2 : (t_2 \dot{\leq} t_3)$. By the IH, either e_1 and e_2 are values or else at least one can be reduced. If either e_1 or e_2 can be reduced, then e can be reduced by applying to corresponding congruence rule. If either e_1 or e_2 is equal to **refl** then **red-trans-refl1** or **red-trans-refl2** applies. If they are both equal to values other than **refl** then we can use Lemma 3.26 to show that they must both have the same top-level constructor and both have the form $\text{funI}(e'_1, e'_2)$ or $\text{subI}(e'_1, e'_2)$ or $\text{allI}(e')e$. In any of those cases one of the **red-trans** reduction rules applies.

□

Theorem 3.28 (Type Soundness). If $\Delta; \cdot \vdash e : t$ then either e is a value or there is some e' such that $e \longrightarrow e'$ and $\Delta; \cdot \vdash e' : t$.

Proof. By Lemmas 3.24 and 3.27. □

Definition 3.29 (Normal Form). A term, e , is in normal form if there is no e' such that $e \longrightarrow e'$.

Definition 3.30 (Strong Normalization). The term e has normalization bound $n \geq 0$, written $e \in \text{SN}_n$, if:

- e is in normal form, or
- there exists a natural number $n' < n$ such that for all e' , $e \longrightarrow e'$ implies $e' \in \text{SN}_{n'}$.

The term e is strongly normalizing, written $e \in \text{SN}$, if $e \in \text{SN}_n$ (for some n). It is clear that if $e \in \text{SN}_n$ then the longest reduction sequence from e has length bounded by n . This provides an induction principle for strongly normalizing terms.

We conjecture that all terms typable in λ_{2C} are strongly normalizing.

Conjecture 3.31 (Strong Normalization). If $\Delta; \cdot \vdash e : t$ then $e \in \text{SN}$.

4 Strong Normalization for λ_{2C^-}

In this section we will prove normalization for λ_{2C^-} , a fragment of λ_{2C} which does not include the subtyping elimination rules (`funEL`, `funER`, `subEL`, `subER`, `allE`). Without the use of subtyping elimination rules, the calculus becomes a bit less flexible. However, many of the important applications of λ_{2C} , such as for encoding GADTs and intentional type analysis, can still be expressed naturally in λ_{2C^-} . Furthermore, it is much easier to prove strong normalization for λ_{2C^-} than for the full λ_{2C} .

It is straightforward to check that each of the lemmas proved about λ_{2C} in the previous section also hold for λ_{2C^-} .

Observe that the only reduction rules that apply to a term whose top-level constructor is an introduction form are congruence rules. Therefore terms consisting of an introduction constructor

applied to strongly normalizing terms are themselves strongly normalizing. For example, if $e_1 \in \text{SN}$ and $e_2 \in \text{SN}$ then $\text{funI}(e_1, e_2) \in \text{SN}$. We will freely make use of this fact.

Lemma 4.1. If for all e' such that $e \longrightarrow e'$ we have $e' \in \text{SN}$, then $e \in \text{SN}$.

Proof. The set of redexes of e is finite, so let $\{e_1, \dots, e_n\} = \{e' \mid e \longrightarrow e'\}$. For such each e_i , fix some n_i , such that $e_i \in \text{SN}_{n_i}$. If we let $n = \max\{n_i\}$ then we have $e \in \text{SN}_n$, so $e \in \text{SN}$. \square

Definition 4.2 (Neutral). A term e is neutral, written $e \downarrow$ if it is not an introduction form. So, for e, e_1, e_2 we have:

$$x \downarrow \quad e_1 e_2 \downarrow \quad \text{coerce}(e_1, e_2) \downarrow \quad \text{trans}(e_1, e_2) \downarrow$$

Definition 4.3 (Reducibility Candidates). A set of terms S is a candidate, written $S \in \mathcal{RC}$, if it satisfies three conditions:

(CR1) $S \subseteq \text{SN}$, and

(CR2) if $e \in S$ and $e \longrightarrow e'$ then $e' \in S$, and

(CR3) if $e \downarrow$ and for all e' such that $e \longrightarrow e'$ we have $e' \in S$, then $e \in S$.

The interpretation of the sort $*$ of types is the set of reducibility candidates.

$$\llbracket * \rrbracket = \mathcal{RC}$$

The set of strongly normalizing terms is a reducibility candidate.

Lemma 4.4. The set SN of strongly normalizing terms is a reducibility candidate, $\text{SN} \in \mathcal{RC}$. We have:

Proof. CR1 and CR2 are trivial, and CR3 is proven by Lemma 4.1. \square

The set of reducibility candidates is closed under non-empty intersections.

Lemma 4.5. If $\{A_i\}_{i \in I}$ is a non-empty family of sets in \mathcal{RC} then $(\bigcap_{i \in I} A_i) \in \mathcal{RC}$.

Proof.

- (CR1) Assume $e \in (\bigcap_{i \in I} A_i)$
 For all $i, e \in A_i$.
 For some $i, e \in A_i$ (because the family is non-empty)
 $e \in \text{SN}$ (by CR1 for A_i)
- (CR2) Assume $e \longrightarrow e'$
 For all $i \in I, e' \in A_i$ (using CR2 for each A_i)
 $e' \in (\bigcap_{i \in I} A_i)$
- (CR3) Assume $e \downarrow$
 Assume for all $e', e \longrightarrow e'$ implies $e' \in (\bigcap_{i \in I} A_i)$
 For all $i \in I, e \in A_i$ (using CR3 for each A_i)
 $e \in (\bigcap_{i \in I} A_i)$

□

We will need to make use of the fact that candidates are non-empty sets.

Lemma 4.6 (Non-empty). For all $A \in \mathcal{RC}$, we have $x \in A$.

Proof. $x \downarrow$ and x has no reductions, so by CR3, $x \in A$. □

Definition 4.7. Given a set of terms A , we define the minimal candidate containing A , $\text{cand}(A)$ by:

- if $e \in A$ then $e \in \text{cand}(A)$,
- if $e \in \text{cand}(A)$ and $e \longrightarrow e'$ then $e' \in \text{cand}(A)$,
- if $e \downarrow$ and for all $e', e \longrightarrow e'$ implies $e' \in \text{cand}(A)$ then $e \in \text{cand}(A)$

Lemma 4.8. For a set of terms $A \subseteq \text{SN}$, $\text{cand}(A) \in \mathcal{RC}$.

Proof. Follows straightforwardly from the definition of $\text{cand}(A)$. □

Definition 4.9. The set of normal neutral terms, NN , is defined by

$$\text{NN} = \{e \mid e \downarrow \text{ and } e \text{ is in normal form}\}$$

The set of partial proofs, PP , is defined by

$$\begin{aligned}
PP_0 &= NN \\
PP_{n+1} &= NN \\
&\cup \{ \text{trans}(e_p, e_s), \text{trans}(e_s, e_p), \text{funI}(e_p, e_s), \text{funI}(e_s, e_p), \\
&\quad \text{subI}(e_p, e_s), \text{subI}(e_s, e_p), \text{allI}(e_p) \mid e_p \in PP_n, e_s \in SN \} \\
PP_\infty &= \bigcup_{i \in \mathbb{N}} PP_i \\
PP &= \text{cand}(PP_\infty)
\end{aligned}$$

Lemma 4.10. If $e_1 \in SN_{n_1}$ and $e_2 \in SN_{n_2}$ then $\text{trans}(e_1, e_2) \in SN$.

Proof.

By induction on simultaneous $n_1 + n_2$ and the structure of e_1 and e_2 . Suppose $\text{trans}(e_1, e_2) \longrightarrow e$. If the rule used is none of `red-trans-refl1`, `red-trans-refl2`, `red-trans-funI`, `red-trans-subI`, or `red-trans-allI`, then $e = \text{trans}(e'_1, e'_2) \in SN$ by IH. Otherwise, suppose its `red-trans-funI`, so $e_1 = \text{funI}(e_{11}, e_{12})$, $e_2 = \text{funI}(e_{21}, e_{22})$ and

$$\text{trans}(\text{funI}(e_{11}, e_{12}), \text{funI}(e_{21}, e_{22})) \longrightarrow \text{funI}(\text{trans}(e_{21}, e_{11}), \text{trans}(e_{12}, e_{22}))$$

It must be that $e_{11}, e_{12} \in SN_{n_1}$ and $e_{21}, e_{22} \in SN_{n_2}$, so by IH, $\text{trans}(e_{21}, e_{11}), \text{trans}(e_{12}, e_{22}) \in SN$, so $\text{funI}(\text{trans}(e_{21}, e_{11}), \text{trans}(e_{12}, e_{22})) \in SN$. The cases for the other reduction rules are similar. \square

Lemma 4.11. $PP_\infty \subseteq SN$.

Proof. Suppose $e \in PP_i$ for some i and we proceed by induction on i .

case: $i = 0$. $e \in NN$ so e is in normal form, so $e \in SN$.

case: $i = j + 1$. If $e \in NN$ then e is in normal form, so $e \in SN$. Otherwise, $e = \text{trans}(e_p, e_s)$ or $e = \text{trans}(e_s, e_p)$ or $e = \text{funI}(e_p, e_s)$ or $e = \text{funI}(e_s, e_p)$ or $e = \text{subI}(e_p, e_s)$ or $e = \text{subI}(e_s, e_p)$ or $e = \text{allI}(e_p)$ for some $e_p \in PP_j, e_s \in SN$. Suppose $e = \text{trans}(e_p, e_s)$. By IH, $e_p \in SN$, so by Lemma 4.10, $\text{trans}(e_p, e_s) \in SN$. Suppose $e = \text{funI}(e_p, e_s)$. By IH, $e_p \in SN$. If $\text{funI}(e_p, e_s) \longrightarrow e$ then $e = \text{funI}(e'_p, e_s)$ and $e_p \longrightarrow e'_p$ or $e = \text{funI}(e_p, e'_s)$ and $e_s \longrightarrow e'_s$, therefore $\text{funI}(e_p, e_s) \in SN$. Each other case for e is similar to the previous \square

Lemma 4.12. For any proof value p , $p \notin \text{PP}$.

Proof. All proof values are in normal form, so it is sufficient to show for all i , $p \notin \text{PP}_i$, by induction on i .

case: $i = 0$. Proof values are normal and not neutral, so they are not in $\text{cand}(\emptyset)$.

case: $i = j + 1$. refl is normal and not neutral, so it is not in PP_i . For each other top-level proof term constructor, all subterms must also be proof values, by IH these proof value subterms are not in PP_j , so $p \notin \text{PP}_i$. \square

Lemma 4.13. If $e_p \in \text{PP}$ and $e_s \in \text{SN}$ then

$$\{\text{funI}(e_p, e_s), \text{funI}(e_s, e_p), \text{subI}(e_p, e_s), \text{subI}(e_s, e_p), \text{allI}(e_p)\} \subseteq \text{PP}$$

Proof. Consider $\text{funI}(e_p, e_s)$. $e_p \in \text{PP}$, so $e_p \in \text{SN}$. If $\text{funI}(e_p, e_s) \longrightarrow e$ then $e = \text{funI}(e'_p, e_s)$ and $e_p \longrightarrow e'_p$ or $e = \text{funI}(e_p, e'_s)$ and $e_s \longrightarrow e'_s$, therefore $\text{funI}(e_p, e_s) \in \text{SN}$. Therefore any reduction path from $\text{funI}(e_p, e_s)$ ends with $\text{funI}(e'_p, e'_s)$ with $e'_p \in \text{PP}_\infty$, $e'_s \in \text{SN}$, therefore $\text{funI}(e_p, e_s) \in \text{PP}$. Each other case is similar. \square

Lemma 4.14. If $e_p \in \text{PP}$ and $e_s \in \text{SN}_{n_s}$ then $\text{trans}(e_p, e_s), \text{trans}(e_s, e_p) \in \text{PP}$.

Proof. $\text{PP} \in \mathcal{RC}$ and $\text{trans}(e_s, e_p) \downarrow$ so we can use CR3. $e_p \in \text{SN}_{n_p}$ by Lemma 4.11, and we proceed by induction on $n_p + n_s$. Suppose $\text{trans}(e_p, e_s) \longrightarrow e$. If the rule used is none of red-trans-refl1 , red-trans-refl2 , red-trans-funI , red-trans-subI , or red-trans-allI , then $e = \text{trans}(e'_p, e'_s) \in \text{PP}$ by IH. Otherwise, suppose its red-trans-funI , so $e_p = \text{funI}(e_{p1}, e_{p2})$, $e_s = \text{funI}(e_{s1}, e_{s2})$ and

$$\text{trans}(\text{funI}(e_{p1}, e_{p2}), \text{funI}(e_{s1}, e_{s2})) \longrightarrow \text{funI}(\text{trans}(e_{s1}, e_{p1}), \text{trans}(e_{p2}, e_{s2}))$$

It must be that $e_{p1}, e_{p2} \in \text{SN}_{n_p}$ and $e_{s1}, e_{s2} \in \text{SN}_{n_s}$, so by IH, $\text{trans}(e_{s1}, e_{p1}), \text{trans}(e_{p2}, e_{s2}) \in \text{PP}$, so $\text{funI}(\text{trans}(e_{p1}, e_{s1}), \text{trans}(e_{p2}, e_{s2})) \in \text{PP}$ by Lemma 4.13. The cases for the other reduction rules and for $\text{trans}(e_s, e_p)$ are similar. \square

Next we will define operators on candidates that correspond to the type constructors.

Definition 4.15. Let A and B be sets of terms and let F_σ be an operator on $[[\sigma]]$. We define the following operations on sets of terms:

$$\begin{aligned} A \rightarrow B &= \{e \mid \text{for all } e' \in A, e e' \in B\} \\ \forall_\sigma F_\sigma &= \bigcap_{X \in [[\sigma]]} F_\sigma(X) \\ (A \dot{\leq} B) &= \begin{cases} \text{SN} & \text{if } A \subseteq B \\ \text{PP} & \text{otherwise} \end{cases} \end{aligned}$$

Lemma 4.16. If $e_1 e_2 \in \text{SN}_n$ then $e_1 \in \text{SN}_n$.

Proof. By induction on n , using the fact that any reduction on e_1 can be transformed into a reduction on $e_1 e_2$ by application of **red-appC1**. \square

Next we will show that the interpretations of type constructors preserve candidates.

Lemma 4.17.

- Let A, B be reducibility candidates ($A \in \mathcal{RC}$ and $B \in \mathcal{RC}$), then $A \rightarrow B \in \mathcal{RC}$

Proof.

(CR1) Assume $e_1 \in A \rightarrow B$

$x \in A$, so $e_1 x \in B$

$e_1 x \in \text{SN}$

(by CR1 for B)

$e_1 \in \text{SN}$

(by Lemma 4.16)

(CR2) Assume $e_1 \in A \rightarrow B$

Assume $e_1 \rightarrow e'_1$

Fix $e_2 \in A$.

Then $e_1 e_2 \in B$.

(by definition of \rightarrow)

$e_1 e_2 \rightarrow e'_1 e_2$

(using **red-appC1**)

$e'_1 e_2 \in B$

(by CR2 for B)

Therefore $e'_1 \in A \rightarrow B$

(by definition of \rightarrow)

(CR3) Assume $e_1 \downarrow$
 Assume for all e'_1 ,
 $e_1 \longrightarrow e'_1$ implies $e'_1 \in A \rightarrow B$
 Fix $e_2 \in A$, with $e_2 \in \text{SN}_n$ (By CR1 for A)
 Show $e_1 e_2 \in B$ by induction on n
 Assume $e_1 e_2 \longrightarrow e$
 Case on last rule in derivation of $e_1 e_2 \longrightarrow e$
 to show $e \in B$:
 [red-appC1] Then $e = e'_1 e_2$ and $e_1 \longrightarrow e'_1$
 So $e'_1 \in A \rightarrow B$ (applying the second assumption)
 $e'_1 e_2 \in B$ (by definition of \rightarrow)
 [red-appC2] Then $e = e_1 e'_2$ and $e_2 \longrightarrow e'_2$
 $e'_2 \in A$ (by CR2 for A)
 There exists $n' < n$ such that $e'_2 \in \text{SN}_{n'}$ (by definition of SN_n)
 $e_1 e'_2 \in B$ (by induction hypothesis)
 Because $e_1 \downarrow$ there are no other cases.
 $e_1 e_2 \in B$ (since $e_1 e_2 \downarrow$, we can apply CR3 for B)
 $e_1 \in A \rightarrow B$ (by definition of \rightarrow)
 □

- Let A, B be reducibility candidates ($A \in \mathcal{RC}$ and $B \in \mathcal{RC}$), then $(A \dot{\leq} B) \in \mathcal{RC}$

Proof. If $A \subseteq B$ then $(A \dot{\leq} B) = \text{SN}$ and $\text{SN} \in \mathcal{RC}$ by Lemma 4.4. If $A \not\subseteq B$ then $(A \dot{\leq} B) = \text{PP}$.
 $\text{PP}_\infty \subseteq \text{SN}$ and $\text{cand}(A) \in \mathcal{RC}$ for any set of terms $A \subseteq \text{SN}$ by Lemma 4.8, so $\text{PP} \in \mathcal{RC}$. □

- If for all $A \in \mathcal{RC}$, $F(A) \in \mathcal{RC}$, then $\forall_* F \in \mathcal{RC}$.

Proof. Directly from Lemma 4.5. □

Definition 4.18.

- A valuation ξ , is a map from StVars to \mathcal{RC} .

- For a valuation ξ , $A \in \mathcal{RC}$ and $\alpha, \beta \in \text{StVars}$ we define:

$$\xi[\alpha \mapsto A](\beta) = \begin{cases} A & \text{if } \alpha = \beta \\ \xi(\beta) & \text{otherwise} \end{cases}$$

Definition 4.19.

- An environment, η , is a map from TmVars to Terms such that $\eta(x) \neq x$ for finitely many x .
- For an environment, η , $e \in \text{Terms}$ and $x, y \in \text{TmVars}$ we define:

$$\eta[x \mapsto e](y) = \begin{cases} e & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

Definition 4.20. Given a valuation ξ , and static term t we define $\llbracket t \rrbracket_\xi \subseteq \text{Terms}$ as follows:

$$\begin{aligned} \llbracket \alpha \rrbracket_\xi &= \xi(\alpha) \\ \llbracket t_1 \rightarrow t_2 \rrbracket_\xi &= \llbracket t_1 \rrbracket_\xi \rightarrow \llbracket t_2 \rrbracket_\xi \\ \llbracket \forall \alpha : \sigma. t \rrbracket_\xi &= \forall \sigma (\lambda (A \in \llbracket \sigma \rrbracket). \llbracket t \rrbracket_{\xi[\alpha \mapsto A]}) \\ &= \bigcap_{A \in \mathcal{RC}} \llbracket t \rrbracket_{\xi[\alpha \mapsto A]} \\ \llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi &= (\llbracket t_1 \rrbracket_\xi \dot{\leq} \llbracket t_2 \rrbracket_\xi) \end{aligned}$$

This definition for interpretations of types is compositional in the following sense.

Lemma 4.21 (Interpretation is Compositional). For all types t, t' and valuations ξ ,

$$\llbracket [t'/\alpha]t \rrbracket_\xi = \llbracket t \rrbracket_{\xi[\alpha \mapsto \llbracket t' \rrbracket_\xi]}$$

Proof. By induction on the structure of t .

case: $t = \alpha$.

$$\llbracket [t'/\alpha]\alpha \rrbracket_\xi = \llbracket t' \rrbracket_\xi = \llbracket \alpha \rrbracket_{\xi[\alpha \mapsto \llbracket t' \rrbracket_\xi]}$$

case: $t = t_1 \rightarrow t_2$.

$$\begin{aligned}
\llbracket [t'/\alpha]t_1 \rightarrow t_2 \rrbracket_\xi &= \llbracket ([t'/\alpha]t_1) \rightarrow ([t'/\alpha]t_2) \rrbracket_\xi \\
&= \llbracket ([t'/\alpha]t_1) \rrbracket_\xi \rightarrow \llbracket ([t'/\alpha]t_2) \rrbracket_\xi \\
&\stackrel{IH}{=} \llbracket t_1 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]} \rightarrow \llbracket t_2 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]} \\
&= \llbracket t_1 \rightarrow t_2 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}
\end{aligned}$$

case: $t = (t_1 \dot{\leq} t_2)$.

$$\llbracket [t'/\alpha](t_1 \dot{\leq} t_2) \rrbracket_\xi = \llbracket ([t'/\alpha]t_1 \dot{\leq} [t'/\alpha]t_2) \rrbracket_\xi$$

If $\llbracket [t'/\alpha]t_1 \rrbracket_\xi \subseteq \llbracket [t'/\alpha]t_2 \rrbracket_\xi$ then by the IH, $\llbracket t_1 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]} \subseteq \llbracket t_2 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}$, so

$$\begin{aligned}
\llbracket ([t'/\alpha]t_1 \dot{\leq} [t'/\alpha]t_2) \rrbracket_\xi &= \text{SN} \\
&= \llbracket (t_1 \dot{\leq} t_2) \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}
\end{aligned}$$

If $\llbracket [t'/\alpha]t_1 \rrbracket_\xi \not\subseteq \llbracket [t'/\alpha]t_2 \rrbracket_\xi$ then by the IH, $\llbracket t_1 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]} \not\subseteq \llbracket t_2 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}$, so

$$\begin{aligned}
\llbracket ([t'/\alpha]t_1 \dot{\leq} [t'/\alpha]t_2) \rrbracket_\xi &= \text{cand}(\emptyset) \\
&= \llbracket (t_1 \dot{\leq} t_2) \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}
\end{aligned}$$

case: $t = (\forall \beta : \sigma.t_1)$. Pick $\beta \in \text{StVars}$ to be fresh for α, t' .

$$\begin{aligned}
\llbracket [t'/\alpha]\forall \beta : \sigma.t_1 \rrbracket_\xi &= \bigcap_{A \in [\sigma]} \llbracket [t'/\alpha]t_1 \rrbracket_{\xi[\beta \mapsto A]} \\
&\stackrel{IH}{=} \bigcap_{A \in [\sigma]} \llbracket t_1 \rrbracket_{\xi[\beta \mapsto A][\alpha \mapsto [t']_\xi[\beta \mapsto A]]} \\
&\stackrel{(a)}{=} \bigcap_{A \in [\sigma]} \llbracket t_1 \rrbracket_{\xi[\beta \mapsto A][\alpha \mapsto [t']_\xi]} \\
&\stackrel{(b)}{=} \bigcap_{A \in [\sigma]} \llbracket t_1 \rrbracket_{\xi[\alpha \mapsto [t']_\xi][\beta \mapsto A]} \\
&= \llbracket \forall \beta : \sigma.t_1 \rrbracket_{\xi[\alpha \mapsto [t']_\xi]}
\end{aligned}$$

Where step (a) is justified because β is fresh for t' and step (b) is justified by the definition of valuation extension and because $\alpha \neq \beta$. \square

Definition 4.22. For a static context Δ , a static term t , a sort σ , we define:

- For a valuation ξ

$$\xi \models \Delta \iff \text{for all } (\alpha : \sigma) \in \Delta, \xi(\alpha) \in \mathcal{RC}$$

- $\Delta \models t : \sigma \iff$ for all ξ such that $\xi \models \Delta$ we have $\llbracket t \rrbracket_\xi \in \mathcal{RC}$.

Definition 4.23. For a context Γ , a term e with free variables in x_1, \dots, x_n , a type t and an environment η , we define:

- $\llbracket e \rrbracket_\eta = [\eta(x_1)/x_1, \dots, \eta(x_n)/x_n]e$,
- $\xi; \eta \models e : t \iff \llbracket e \rrbracket_\eta \in \llbracket t \rrbracket_\xi$,
- $\xi; \eta \models \Gamma \iff$ for all $(x : t) \in \Gamma, \xi; \eta \models x : t$,
- $\Delta; \Gamma \models e : t \iff$ for all η, ξ such that $\xi \models \Delta$ and $\xi; \eta \models \Gamma$, we have $\xi; \eta \models e : t$.

Lemma 4.24. If $\mathcal{D} :: \Delta \vdash t : \sigma$ then $\Delta \models t : \sigma$.

Proof. By induction on the structure of \mathcal{D} , making use of Lemma 4.17. □

Lemma 4.25. If $\mathcal{D} :: e \longrightarrow e'$ then $[e_1/x]e \longrightarrow [e_1/x]e'$

Proof. By induction on the structure of \mathcal{D} . □

The next few lemmas show that the introduction rules are sound in this interpretation of typing derivations.

Lemma 4.26. Let $(\lambda x.e), e'$ be terms and $A \in \mathcal{RC}$ and $B \in \mathcal{RC}$ and suppose $e' \in A$, $e' \in \text{SN}_n$ and for all $e'' \in A$, $[e''/x]e \in B$, then $(\lambda x.e) e' \in B$.

Proof. By induction on n .

We proceed by using CR3 for B . We have $(\lambda x.e) e' \downarrow$, so we just need to show for all e'' , $(\lambda x.e) e' \longrightarrow e''$ implies $e'' \in B$.

Case on the last rule in the derivation of $(\lambda x.e) e' \longrightarrow e''$

[red-appC1] Then $e'' = (\lambda x.e''') e'$ with $e \longrightarrow e'''$

For all $e_1 \in A$, $[e_1/x]e''' \in B$ (by CR1 for B)

$(\lambda x.e''') e' \in A \rightarrow B$. (by IH)

[red-appC2] Then $e'' = (\lambda x.e) e'''$ with $e' \longrightarrow e'''$

For some $n' < n$, $e''' \in \text{SN}_{n'}$ (by definition of SN_n)

$(\lambda x.e) e''' \in B$ (by IH)

[red-app] Then $e'' = [e'/x]e$

$[e'/x]e \in B$ (by applying assumption)

□

Lemma 4.27 (Soundness of abs). Given

- $\xi \models \Delta$,
- $\xi; \eta \models \Gamma$,
- $\Delta \vdash t_1 : *$ and $\Delta \vdash t_2 : *$,

if for all terms $e_1 \in \llbracket t_1 \rrbracket_\xi$, $\xi; \eta[x \mapsto e_1] \models e : t_2$ then $\xi; \eta \models \lambda x.e : t_1 \rightarrow t_2$.

Proof. $\llbracket \lambda x.e \rrbracket_\eta$ is a lambda term, so it is in normal form, so $\llbracket \lambda x.e \rrbracket_\eta \in \text{SN}$. We just need to show, for all $e' \in \llbracket t_1 \rrbracket_\xi$, we have $\llbracket \lambda x.e \rrbracket_\eta e' \in \llbracket t_2 \rrbracket_\xi$.

$\llbracket t_2 \rrbracket_\xi \in \mathcal{RC}$ (by Lemma 4.24)

Fix $e' \in \llbracket t_1 \rrbracket_\xi$, show $\llbracket \lambda x.e \rrbracket_\eta e' \in \llbracket t_2 \rrbracket_\xi$

$\llbracket t_1 \rrbracket_\xi \in \mathcal{RC}$ and $\llbracket t_2 \rrbracket_\xi \in \mathcal{RC}$ (by Lemma 4.24)

For some n , $e' \in \text{SN}_n$ (by CR1 for $\llbracket t_1 \rrbracket_\xi$)

$\llbracket \lambda x.e \rrbracket_\eta e' \in \llbracket t_2 \rrbracket_\xi$ (by Lemma 4.26)

$\llbracket \lambda x.e \rrbracket_\eta \in \llbracket t_1 \rrbracket_\xi \rightarrow \llbracket t_2 \rrbracket_\xi$ (by definition of \rightarrow)

$\llbracket \lambda x.e \rrbracket_\eta \in \llbracket t_1 \rightarrow t_2 \rrbracket_\xi$ (by definition of $\llbracket t_1 \rightarrow t_2 \rrbracket_\xi$)

$\xi; \eta \models \lambda x.e : t_1 \rightarrow t_2$

□

Lemma 4.28 (Soundness of Subtyping Introductions). Given

- $\xi \models \Delta$, and

• $\xi; \eta \models \Gamma$,

1. if $\Delta \vdash (t'_1 \dot{\leq} t_1) : *$, $\Delta \vdash (t_2 \dot{\leq} t'_2) : *$, $\xi; \eta \models e_1 : (t'_1 \dot{\leq} t_1)$ and $\xi; \eta \models e_2 : (t_2 \dot{\leq} t'_2)$ then

$$\xi; \eta \models \text{funI}(e_1, e_2) : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)$$

and

$$\xi; \eta \models \text{subI}(e_1, e_2) : ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))$$

Proof for funI(e_1, e_2).

Suppose $\llbracket t'_1 \rrbracket_\xi \subseteq \llbracket t_1 \rrbracket_\xi$, and $\llbracket t_2 \rrbracket_\xi \subseteq \llbracket t'_2 \rrbracket_\xi$

$$\llbracket (t'_1 \dot{\leq} t_1) \rrbracket_\xi, \llbracket (t_2 \dot{\leq} t'_2) \rrbracket_\xi \in \mathcal{RC} \quad (\text{by Lemma 4.24})$$

$$\llbracket e_1 \rrbracket_\eta, \llbracket e_2 \rrbracket_\eta \in \text{SN} \quad (\text{by CR1})$$

$$\llbracket \text{funI}(e_1, e_2) \rrbracket_\eta \in \text{SN}$$

$$\text{Need: } \llbracket t_1 \rightarrow t_2 \rrbracket_\xi \subseteq \llbracket t'_1 \rightarrow t'_2 \rrbracket_\xi$$

Fix $e \in \llbracket t_1 \rightarrow t_2 \rrbracket_\xi$,

then for all $e' \in \llbracket t_1 \rrbracket_\xi$, $e e' \in \llbracket t_2 \rrbracket_\xi$

$$\llbracket t_1 \rightarrow t_2 \rrbracket_\xi \in \mathcal{RC} \quad (\text{by Lemma 4.24})$$

$$e \in \text{SN} \quad (\text{by CR1})$$

Need: for all $e' \in \llbracket t'_1 \rrbracket_\xi$, $e e' \in \llbracket t'_2 \rrbracket_\xi$.

Fix $e' \in \llbracket t'_1 \rrbracket_\xi$,

$$\text{then } e' \in \llbracket t_1 \rrbracket_\xi \quad (\text{by } \llbracket t'_1 \rrbracket_\xi \subseteq \llbracket t_1 \rrbracket_\xi)$$

$$e e' \in \llbracket t_2 \rrbracket_\xi$$

$$e e' \in \llbracket t'_2 \rrbracket_\xi \quad (\text{by } \llbracket t_2 \rrbracket_\xi \subseteq \llbracket t'_2 \rrbracket_\xi)$$

Suppose $\llbracket t'_1 \rrbracket_\xi \not\subseteq \llbracket t_1 \rrbracket_\xi$.

Then $\llbracket e_1 \rrbracket_\eta \in \text{PP}$. (by definition of $(\dot{\leq})$)

$\llbracket e_2 \rrbracket_\eta \in \text{SN}$ (by CR1)

$\llbracket \text{funI}(e_1, e_2) \rrbracket_\eta \in \text{PP} \subseteq \text{SN}$ (by Lemmas 4.13 and 4.11)

$\llbracket \text{funI}(e_1, e_2) \rrbracket_\eta \in \llbracket (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2) \rrbracket_\xi$ (by definition of $(\dot{\leq})$)

The case for $\llbracket t_2 \rrbracket_\xi \not\subseteq \llbracket t'_2 \rrbracket_\xi$ is similar.

The proof for subI(e_1, e_2) is essentially the same. □

2. if for all $A \in \llbracket \sigma \rrbracket$, $\xi[\alpha \mapsto A]; \eta \models e : (t \dot{\leq} t')$ then

$$\xi; \eta \models \text{allI}(e) : (\forall \alpha : \sigma. t \dot{\leq} \forall \alpha : \sigma. t')$$

Proof. We have $\llbracket e \rrbracket_\eta \in \text{SN}$. If for each $A \in \llbracket \sigma \rrbracket$, $\llbracket t \rrbracket_{\xi[\alpha \mapsto A]} \subseteq \llbracket t' \rrbracket_{\xi[\alpha \mapsto A]}$, then $\llbracket \text{allI}(e) \rrbracket_\eta \in \text{SN}$ and

$$\bigcap_{A \in \llbracket \sigma \rrbracket} \llbracket t \rrbracket_{\xi[\alpha \mapsto A]} \subseteq \bigcap_{A \in \llbracket \sigma \rrbracket} \llbracket t' \rrbracket_{\xi[\alpha \mapsto A]}$$

Otherwise, for some $A \in \llbracket \sigma \rrbracket$, $\llbracket t \rrbracket_{\xi[\alpha \mapsto A]} \not\subseteq \llbracket t' \rrbracket_{\xi[\alpha \mapsto A]}$. Therefore, by definition of $(\dot{\leq})$, $\llbracket e \rrbracket_\eta \in \text{PP}$ and by Lemma 4.13, $\llbracket \text{allI}(e) \rrbracket_\eta \in \text{PP} \subseteq \llbracket (\forall \alpha : \sigma. t \dot{\leq} \forall \alpha : \sigma. t') \rrbracket_\xi$. \square

Lemma 4.29. Given

- $e_1 \in \text{SN}_{n_1}$, $e_2 \in \text{SN}_{n_2}$,
- $e_2 \in \llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi$,

if $e_1 \in \llbracket t_1 \rrbracket_\xi$ then $\text{coerce}(e_1, e_2) \in \llbracket t_2 \rrbracket_\xi$.

Proof. By induction on $n_1 + n_2$. We proceed by applying CR3 for $\llbracket t_2 \rrbracket_\xi$. We have $\text{coerce}(e_1, e_2) \downarrow$, so we just need to show that for all e' , $\text{coerce}(e_1, e_2) \longrightarrow e'$ implies $e' \in \llbracket t_2 \rrbracket_\xi$. There are three possible rules which can derive $\text{coerce}(e_1, e_2) \longrightarrow e'$, namely **red-coerce**, **red-coerceC1** and **red-coerceC2**.

case: red-coerce. Then $e' = e_1 \in \llbracket t_1 \rrbracket_\xi$, and e_2 is a proof value, so by Lemma 4.12, $e_2 \notin \text{PP}$, so by the definition of $\llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi$ we have $\llbracket t_1 \rrbracket_\xi \subseteq \llbracket t_2 \rrbracket_\xi$, so $e_1 \in \llbracket t_2 \rrbracket_\xi$.

case: red-coerceC1. Then $e' = \text{coerce}(e'_1, e_2)$ and for some $n'_1 < n_1$, $e'_1 \in \text{SN}_{n'_1}$. By CR2 for $\llbracket t_1 \rrbracket_\xi$ we have $e'_1 \in \llbracket t_1 \rrbracket_\xi$. By the IH we have $\text{coerce}(e'_1, e_2) \in \llbracket t_2 \rrbracket_\xi$.

case: red-coerceC2. Then $e' = \text{coerce}(e_1, e'_2)$ and for some $n'_2 < n_2$, $e'_2 \in \text{SN}_{n'_2}$. By CR2 for $\llbracket t_2 \rrbracket_\xi$, $e'_2 \in \llbracket t_2 \rrbracket_\xi$. By the IH we have $\text{coerce}(e_1, e'_2) \in \llbracket t_2 \rrbracket_\xi$. \square

Lemma 4.30 (Soundness of coerce). Given

- $\xi \models \Delta$,
- $\xi; \eta \models \Gamma$,
- $\Delta \vdash t_1 : *$ and $\Delta \vdash (t_1 \dot{\leq} t_2) : *$,

if $\xi; \eta \models e_1 : t_1$ and $\xi; \eta \models e_2 : (t_1 \dot{\leq} t_2)$ then $\xi; \eta \models e_1 : t_2$.

Proof. By Lemma 4.24 we have $\llbracket t_1 \rrbracket_\xi \in \mathcal{RC}$ and $\llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi \in \mathcal{RC}$, and by CR1 we have $\llbracket e_1 \rrbracket_\eta \in \text{SN}$ and $\llbracket e_2 \rrbracket_\eta \in \text{SN}$. By Lemma 4.29 $\llbracket \text{coerce}(e_1, e_2) \rrbracket_\eta \in \llbracket t_2 \rrbracket_\xi$. \square

Lemma 4.31 (Soundness of trans). Given

- $\xi \models \Delta$,
- $\xi; \eta \models \Gamma$,
- $\Delta \vdash (t_1 \dot{\leq} t_2) : *$, and $\Delta \vdash (t_2 \dot{\leq} t_3) : *$,

if $\xi; \eta \models e_1 : (t_1 \dot{\leq} t_2)$ and $\xi; \eta \models e_2 : (t_2 \dot{\leq} t_3)$ then $\xi; \eta \models \text{trans}(e_1, e_2) : (t_1 \dot{\leq} t_3)$.

Proof.

Suppose $\llbracket t_1 \rrbracket_\xi \subseteq \llbracket t_2 \rrbracket_\xi$ and $\llbracket t_2 \rrbracket_\xi \subseteq \llbracket t_3 \rrbracket_\xi$. Then $\llbracket t_1 \rrbracket_\xi \subseteq \llbracket t_3 \rrbracket_\xi$

$\llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi, \llbracket (t_2 \dot{\leq} t_3) \rrbracket_\xi \in \mathcal{RC}$ (by Lemma 4.24)

$\llbracket e_1 \rrbracket_\eta, \llbracket e_2 \rrbracket_\eta \in \text{SN}$ (by CR1)

$\llbracket \text{trans}(e_1, e_2) \rrbracket_\eta \in \text{SN}$ (by Lemma 4.10)

$\llbracket \text{trans}(e_1, e_2) \rrbracket_\eta \in \llbracket (t_1 \dot{\leq} t_3) \rrbracket_\xi$

Suppose $\llbracket t_1 \rrbracket_\xi \not\subseteq \llbracket t_2 \rrbracket_\xi$

Then $\llbracket e_1 \rrbracket_\eta \in \text{PP}$ (by definition of $(\dot{\leq})$)

$\llbracket e_2 \rrbracket_\eta \in \text{SN}$ (by CR1)

$\llbracket \text{trans}(e_1, e_2) \rrbracket_\eta \in \llbracket (t_1 \dot{\leq} t_3) \rrbracket_\xi$ (by Lemma 4.14)

The case of $\llbracket t_2 \rrbracket_\xi \not\subseteq \llbracket t_3 \rrbracket_\xi$ is similar \square

Lemma 4.32. If $\Delta; \Gamma \models e : t$ and there exists ξ, η such that $\xi \models \Delta$ and $\xi; \eta \models \Gamma$, then $\llbracket e \rrbracket_\eta \in \text{SN}$.

Proof. We have $\xi; \eta \models e : t$, so $\llbracket e \rrbracket_\eta \in \llbracket t \rrbracket_\xi$. By Lemmas 4.24 and CR1 for $\llbracket t \rrbracket_\xi$, we have $\llbracket e \rrbracket_\eta \in \text{SN}$. \square

Lemma 4.33 (Main Lemma). If $\Delta; \Gamma \vdash e : t$ then $\Delta; \Gamma \models e : t$.

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : t$.

Fix any ξ, η such that $\xi \models \Delta$ and $\xi; \eta \models \Gamma$

case: var. $\Delta; \Gamma \vdash e : t$ where $e = x$ and $x : t \in \Gamma$.

$\eta(x) \in \llbracket t \rrbracket_\xi$ (by definition of $\xi; \eta \models \Gamma$)

case: abs. $\Delta; \Gamma \vdash e : t$ where $t = t_1 \rightarrow t_2$, $e = \lambda x.e'$, $\Delta \vdash t_1 : *$ and $\Delta; \Gamma, x : t_1 \vdash e' : t_2$.

For all $e_1 \in \llbracket t_1 \rrbracket_\xi$,

$$\xi; \eta[x \mapsto e_1] \models e' : t_2 \quad (\text{by applying induction hypothesis})$$

$$\Delta \vdash t_2 : * \quad (\text{by Lemma 3.16})$$

$$\xi; \eta \models \lambda x.e' : t_1 \rightarrow t_2 \quad (\text{by Lemma 4.27})$$

case: app. $\Delta; \Gamma \vdash e : t$ where $e = e_1 e_2$ and $\Delta; \Gamma \vdash e_1 : t_1 \rightarrow t_2$ and $\Delta; \Gamma \vdash e_2 : t_1$.

$$\xi; \eta \models e_1 : t_1 \rightarrow t_2 \text{ and } \xi; \eta \models e_2 : t_1 \quad (\text{by applying induction hypothesis})$$

$$\xi; \eta \models e_1 e_2 : t_2 \quad (\text{by definition of } \rightarrow)$$

case: tabs. $\Delta; \Gamma \vdash e : t$ where $t = \forall \alpha : \sigma.t'$ and $\Delta, \alpha : \sigma; \Gamma \vdash e : t'$.

$$\text{For all } A \in \llbracket \sigma \rrbracket, \xi[\alpha \mapsto A]; \eta \models e : t' \quad (\text{by induction hypothesis})$$

$$\xi; \eta \models e : \forall \alpha : \sigma.t' \quad (\text{by semantics of } \forall)$$

case: tapp. $\Delta; \Gamma \vdash e : t$ where $t = [t_1/\alpha]t_2$ and $\Delta; \Gamma \vdash e : \forall \alpha : \sigma.t_2$ and $\Delta \vdash t_1 : \sigma$.

$$\xi; \eta \models e : \forall \alpha : \sigma.t_2 \quad (\text{by applying induction hypothesis})$$

$$\text{For all } A \in \llbracket \sigma \rrbracket, \xi[\alpha \mapsto A]; \eta \models e : t_2 \quad (\text{by semantics of } \forall)$$

$$\xi[\alpha \mapsto \llbracket t_1 \rrbracket_\xi]; \eta \models e : t_2 \quad (\text{by instantiating previous line})$$

$$\xi; \eta \models e : [t_1/\alpha]t_2 \quad (\text{by Lemma 4.21})$$

case: coerce. $\Delta; \Gamma \vdash e : t$ where $e = \text{coerce}(e_1, e_2)$, $\Delta; \Gamma \vdash e_1 : t_1$ and $\Delta; \Gamma \vdash e_2 : (t_1 \dot{\leq} t)$.

$$\xi; \eta \models e_1 : t_1 \quad (\text{by IH})$$

$$\xi; \eta \models e_2 : (t_1 \dot{\leq} t) \quad (\text{by IH})$$

$$\Delta \vdash t_1 : * \text{ and } \Delta \vdash (t_1 \dot{\leq} t) : * \quad (\text{by Lemma 3.16})$$

$$\xi; \eta \models \text{coerce}(e_1, e_2) : t \quad (\text{by Lemma 4.30})$$

case: funl. $\Delta; \Gamma \vdash e : t$ where $t = (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2)$, $e = \text{funI}(e_1, e_2)$, $\Delta; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1)$ and $\Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)$.

$$\xi; \eta \models e_1 : (t'_1 \dot{\leq} t_1) \quad (\text{by IH})$$

$$\xi; \eta \models e_2 : (t_2 \dot{\leq} t'_2) \quad (\text{by IH})$$

$$\Delta \vdash (t'_1 \dot{\leq} t_1) \text{ and } \Delta \vdash (t_2 \dot{\leq} t'_2) \quad (\text{by Lemma 3.16})$$

$$\xi; \eta \models \text{funI}(e_1, e_2) : (t_1 \rightarrow t_2 \dot{\leq} t'_1 \rightarrow t'_2) \quad (\text{by Lemma 4.28})$$

case: subl. $\Delta; \Gamma \vdash e : t$ where $t = ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))$, $e = \text{subI}(e_1, e_2)$, $\Delta; \Gamma \vdash e_1 : (t'_1 \dot{\leq} t_1)$ and $\Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)$.

$\xi; \eta \models e_1 : (t'_1 \dot{\leq} t_1)$ (by IH)

$\xi; \eta \models e_2 : (t_2 \dot{\leq} t'_2)$ (by IH)

$\Delta \vdash (t'_1 \dot{\leq} t_1)$ and $\Delta \vdash (t_2 \dot{\leq} t'_2)$ (by Lemma 3.16)

$\xi; \eta \models \text{subI}(e_1, e_2) : ((t_1 \dot{\leq} t_2) \dot{\leq} (t'_1 \dot{\leq} t'_2))$ (by Lemma 4.28)

case: allI. $\Delta; \Gamma \vdash e : t$ where $t = (\forall \alpha : \sigma. t_1 \dot{\leq} \forall \alpha : \sigma. t'_1)$, $e = \text{allI}(e_1)$ and $\Delta, \alpha : \sigma; \Gamma \vdash e_1 : (t_1 \dot{\leq} t'_1)$.

For all $A \in \llbracket \sigma \rrbracket$, $\xi[\alpha \mapsto A]; \eta \models e_1 : (t_1 \dot{\leq} t'_1)$ (by IH)

$\xi; \eta \models \text{allI}(e_1) : (\forall \alpha : \sigma. t_1 \dot{\leq} \forall \alpha : \sigma. t'_1)$ (by Lemma 4.28)

case: trans. $\Delta; \Gamma \vdash e : t$ where $t = (t_1 \dot{\leq} t_3)$, $e = \text{trans}(e_1, e_2)$, $\Delta, \alpha : \sigma; \Gamma \vdash e_1 : (t_1 \dot{\leq} t_2)$, and $\Delta, \alpha : \sigma; \Gamma \vdash e_2 : (t_2 \dot{\leq} t_3)$.

$\xi; \eta \models e_1 : (t_1 \dot{\leq} t_2)$ (by IH)

$\xi; \eta \models e_2 : (t_2 \dot{\leq} t_3)$ (by IH)

$\Delta \vdash (t_1 \dot{\leq} t_2)$ and $\Delta \vdash (t_2 \dot{\leq} t_3)$ (by Lemma 3.16)

$\xi; \eta \models \text{trans}(e_1, e_2) : (t_1 \dot{\leq} t_3)$ (by Lemma 4.31)

□

Theorem 4.34. If $\Delta; \Gamma \vdash e : t$, $\xi \models \Delta$ and $\xi; \eta \models \Gamma$ then $\llbracket e \rrbracket_\eta \in \text{SN}$.

Proof. Immediate consequence of Lemmas 4.32 and 4.33. □

Since the the set of reducibility candidates is non-empty, for any Δ there exists ξ such that $\xi \models \Delta$. So as a corollary of the theorem we get.

Corollary 4.35. If $\Delta; \cdot \vdash e : t$ then $e \in \text{SN}$.

Proof. Pick ξ such that $\xi(\alpha) = \text{SN}$ for $\alpha \in \text{dom}(\Delta)$, then $\xi \models \Delta$. For any η , $\xi; \eta \models \cdot$, so the conclusion follows from the previous theorem. □

We can then use strong normalization, along with the progress and preservation lemmas, to prove that λ_{2C} - corresponds to a consistent logic. To do this, we just need to show that there is some unprovable proposition.

Lemma 4.36. For any $n \geq 0$, there is no e such that $\Delta; \cdot \vdash e : \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. \forall \alpha. \alpha$.

Proof. Assume $\mathcal{D} :: \Delta; \cdot \vdash e : \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. \forall \alpha. \alpha$. Then by the previous corollary, $e \in \text{SN}_m$, and we proceed to prove a contradiction by induction on m with a nested induction on the structure

of \mathcal{D} . If $e \longrightarrow e'$ then by Lemma 3.24 (preservation) $\Delta \cdot \vdash e' : \forall \alpha_1 : \sigma_1 \dots \forall \alpha_n : \sigma_n. \forall \alpha. \alpha$ and by the IH we have a contradiction. Otherwise, e is in normal form and by Lemma 3.27 (progress) e must be a generalized value. Consider each of the possible last rules used in \mathcal{D} . Because e is a generalized value, the only possibilities are **tapp** or **tabs**. In the **tapp** case we can apply the IH to get a contradiction. Otherwise, we must have the last rule must be **tabs**. If $n > 0$, then we can apply the IH to get a contradiction. Otherwise we have $\Delta, \alpha : \sigma; \cdot \vdash e : \alpha$. Consider the possible last rules for this derivation. Because e is a generalized value, the only possible rule is **tapp**, but then we can apply the IH to get a contradiction. \square

Corollary 4.37. There is no e such that $\Delta; \cdot \vdash e : \forall \alpha. \alpha$.

5 $\lambda_{2C^-}^{(), \times, +}$: An Extension of λ_{2C^-}

5.1 Definition of $\lambda_{2C^-}^{(), \times, +}$

$\lambda_{2C^-}^{(), \times, +}$ is an extension of λ_{2C^-} by unit, product and sum types and associated subtyping constraint introduction and elimination forms. We will show that this extension preserves strong normalization. It is necessary to add unit, product and sum types into the calculus rather than use impredicative encodings because the lack of subtyping elimination rules for \rightarrow and \forall mean that impredicative encodings do not have subtyping elimination rules. The syntax of $\lambda_{2C^-}^{(), \times, +}$ is the same as for λ_{2C^-} with the following additions:

$$\begin{aligned}
 \text{(static terms)} \quad t & := \dots \mid () \mid t_1 \times t_2 \mid t_1 + t_2 \\
 \text{(terms)} \quad e & := \dots \mid () \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{inl}(e) \mid \text{inr}(e) \\
 & \quad \mid \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \mid \text{unitI} \mid \text{prodI}(e_1, e_2) \mid \text{prodEL}(e) \\
 & \quad \mid \text{prodER}(e) \mid \text{sumI}(e_1, e_2) \mid \text{sumEL}(e) \mid \text{sumER}(e)
 \end{aligned}$$

Definition 5.1 (Proof Values). The proof values of $\lambda_{2C^-}^{(), \times, +}$ are a superset of the proof values of λ_{2C^-} .

$$\text{(proof values)} \quad p := \dots \mid \text{unitI} \mid \text{prodI}(p_1, p_2) \mid \text{sumI}(p_1, p_2)$$

Definition 5.2 (Reduction). The reduction rules of $\lambda_{2C^-}^{(), \times, +}$ includes all the reduction rules of λ_{2C^-} .

In addition there are the following reduction rules:

$$\begin{array}{c}
\frac{}{\text{fst}(e_1, e_2) \longrightarrow e_1} \text{red-fst} \qquad \frac{}{\text{snd}(e_1, e_2) \longrightarrow e_2} \text{red-snd} \\
\\
\frac{}{\text{case inl}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow [e/x]e_1} \text{red-case1} \\
\frac{}{\text{case inr}(e) \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow [e/y]e_2} \text{red-case2} \\
\\
\frac{}{\text{prodEL}(\text{prodI}(e_1, e_2)) \longrightarrow e_1} \text{red-prod1} \qquad \frac{}{\text{prodER}(\text{prodI}(e_1, e_2)) \longrightarrow e_2} \text{red-prod2} \\
\frac{}{\text{sumEL}(\text{sumI}(e_1, e_2)) \longrightarrow e_1} \text{red-sum1} \qquad \frac{}{\text{sumER}(\text{sumI}(e_1, e_2)) \longrightarrow e_2} \text{red-sum2}
\end{array}$$

The following new congruence rules are added:

$$\begin{array}{c}
\frac{e \longrightarrow e'}{\text{fst } e \longrightarrow \text{fst } e'} \text{red-fstC} \qquad \frac{e \longrightarrow e'}{\text{snd } e \longrightarrow \text{snd } e'} \text{red-sndC} \\
\\
\frac{e_1 \longrightarrow e'_1}{(e_1, e_2) \longrightarrow (e'_1, e_2)} \text{red-pairC1} \qquad \frac{e_2 \longrightarrow e'_2}{(e_1, e_2) \longrightarrow (e_1, e'_2)} \text{red-pairC2} \\
\\
\frac{e \longrightarrow e'}{\text{inl}(e) \longrightarrow \text{inl}(e')} \text{red-inlC} \qquad \frac{e \longrightarrow e'}{\text{inr}(e) \longrightarrow \text{inr}(e')} \text{red-inrC} \\
\\
\frac{e \longrightarrow e'}{\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow \text{case } e' \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}} \text{red-caseC1} \\
\frac{e_1 \longrightarrow e'_1}{\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e'_1, \text{inr}(y) \rightarrow e_2\}} \text{red-caseC2} \\
\frac{e_2 \longrightarrow e'_2}{\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e'_2\}} \text{red-caseC3} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{prodI}(e_1, e_2) \longrightarrow \text{prodI}(e'_1, e_2)} \text{red-prodIC1} \qquad \frac{e_2 \longrightarrow e'_2}{\text{prodI}(e_1, e_2) \longrightarrow \text{prodI}(e_1, e'_2)} \text{red-prodIC1} \\
\frac{e \longrightarrow e'}{\text{prodEL}(e) \longrightarrow \text{prodEL}(e')} \text{red-prodELC} \qquad \frac{e \longrightarrow e'}{\text{prodER}(e) \longrightarrow \text{prodER}(e')} \text{red-prodERC} \\
\\
\frac{e_1 \longrightarrow e'_1}{\text{sumI}(e_1, e_2) \longrightarrow \text{sumI}(e'_1, e_2)} \text{red-sumIC1} \qquad \frac{e_2 \longrightarrow e'_2}{\text{sumI}(e_1, e_2) \longrightarrow \text{sumI}(e_1, e'_2)} \text{red-sumIC1} \\
\frac{e \longrightarrow e'}{\text{sumEL}(e) \longrightarrow \text{sumEL}(e')} \text{red-sumELC} \qquad \frac{e \longrightarrow e'}{\text{sumER}(e) \longrightarrow \text{sumER}(e')} \text{red-sumERC}
\end{array}$$

The following rules deal with eliminations applied to reflexivity and with transitivity.

$$\begin{array}{c}
\frac{}{\text{prodEL}(\text{refl}) \longrightarrow \text{refl}} \text{red-prodELrefl} \quad \frac{}{\text{prodEL}(\text{refl}) \longrightarrow \text{refl}} \text{red-prodERrefl} \\
\\
\frac{}{\text{sumEL}(\text{refl}) \longrightarrow \text{refl}} \text{red-sumELrefl} \quad \frac{}{\text{sumEL}(\text{refl}) \longrightarrow \text{refl}} \text{red-sumERrefl} \\
\\
\frac{}{\text{trans}(\text{prodI}(e_1, e_2), \text{prodI}(e'_1, e'_2)) \longrightarrow \text{prodI}(\text{trans}(e_1, e'_1), \text{trans}(e_2, e'_2))} \text{red-trans-prodI} \\
\\
\frac{}{\text{trans}(\text{sumI}(e_1, e_2), \text{sumI}(e'_1, e'_2)) \longrightarrow \text{sumI}(\text{trans}(e_1, e'_1), \text{trans}(e_2, e'_2))} \text{red-trans-sumI} \\
\\
\frac{}{\text{trans}(\text{unitI}, \text{unitI}) \longrightarrow \text{unitI}} \text{red-trans-unitI}
\end{array}$$

Definition 5.3 (Sorting). The sorting rules of $\lambda_{2C^-}^{(), \times, +}$ include all those of λ_{2C^-} plus the following:

$$\begin{array}{c}
\frac{}{\Delta \vdash () : *} \text{sort-unit} \quad \frac{\Delta \vdash t_1 : * \quad \Delta \vdash t_2 : *}{\Delta \vdash t_1 \times t_2 : *} \text{sort-prod} \\
\\
\frac{\Delta \vdash t_1 : * \quad \Delta \vdash t_2 : *}{\Delta \vdash t_1 + t_2 : *} \text{sort-sum}
\end{array}$$

Definition 5.4 (Type Assignment). The typing rules of $\lambda_{2C^-}^{(), \times, +}$ include all those of λ_{2C^-} plus the following:

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash () : ()} \text{unit} \quad \frac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta; \Gamma \vdash e_2 : t_2}{\Delta; \Gamma \vdash (e_1, e_2) : t_1 \times t_2} \text{pair} \\
\\
\frac{\Delta; \Gamma \vdash e : t_1 \times t_2}{\Delta; \Gamma \vdash \text{fst } e : t_1} \text{fst} \quad \frac{\Delta; \Gamma \vdash e : t_1 \times t_2}{\Delta; \Gamma \vdash \text{snd } e : t_2} \text{snd} \\
\\
\frac{\Delta; \Gamma \vdash e : t_1}{\Delta; \Gamma \vdash \text{inl}(e) : t_1 + t_2} \text{inl} \quad \frac{\Delta; \Gamma \vdash e : t_2}{\Delta; \Gamma \vdash \text{inr}(e) : t_2 + t_2} \text{inr} \\
\\
\frac{\Delta; \Gamma \vdash e : t_1 + t_2 \quad \Delta; \Gamma, x : t_1 \vdash e_1 : t \quad \Delta; \Gamma, y : t_2 \vdash e_2 : t}{\Delta; \Gamma \vdash \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} : t} \text{case} \\
\\
\frac{\Delta; \Gamma \vdash e_1 : (t_1 \dot{\leq} t'_1) \quad \Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta; \Gamma \vdash \text{prodI}(e_1, e_2) : (t_1 \times t_2 \dot{\leq} t'_1 \times t'_2)} \text{red-prodI} \\
\\
\frac{\Delta; \Gamma \vdash e : (t_1 \times t_2 \dot{\leq} t'_1 \times t'_2)}{\Delta; \Gamma \vdash \text{prodEL}(e) : (t_1 \dot{\leq} t'_1)} \text{red-prodEL} \quad \frac{\Delta; \Gamma \vdash e : (t_1 \times t_2 \dot{\leq} t'_1 \times t'_2)}{\Delta; \Gamma \vdash \text{prodER}(e) : (t_2 \dot{\leq} t'_2)} \text{red-prodER} \\
\\
\frac{\Delta; \Gamma \vdash e_1 : (t_1 \dot{\leq} t'_1) \quad \Delta; \Gamma \vdash e_2 : (t_2 \dot{\leq} t'_2)}{\Delta; \Gamma \vdash \text{sumI}(e_1, e_2) : (t_1 + t_2 \dot{\leq} t'_1 + t'_2)} \text{red-sumI} \\
\\
\frac{\Delta; \Gamma \vdash e : (t_1 + t_2 \dot{\leq} t'_1 + t'_2)}{\Delta; \Gamma \vdash \text{sumEL}(e) : (t_1 \dot{\leq} t'_1)} \text{red-sumEL} \quad \frac{\Delta; \Gamma \vdash e : (t_1 + t_2 \dot{\leq} t'_1 + t'_2)}{\Delta; \Gamma \vdash \text{sumER}(e) : (t_2 \dot{\leq} t'_2)} \text{red-sumER}
\end{array}$$

Each of the Lemmas proved of λ_{2C} in the first section can be extended straightforwardly to $\lambda_{2C}^{(0),\times,+}$. We will only state type soundness and its key Lemmas.

Definition 5.5 (Values). The (generalized) values of $\lambda_{2C}^{(0),\times,+}$ are a superset of those of λ_{2C} :

$$\text{(values)} \quad v \quad := \quad \dots \mid () \mid (v_1, v_2) \mid \text{inl}(v) \mid \text{inr}(v) \mid \text{unitI} \mid \text{prodI}(v_1, v_2) \mid \text{sumI}(v_1, v_2)$$

Lemma 5.6 (Preservation). If $\Delta; \Gamma \vdash e : t$ and $e \longrightarrow e'$ then $\Delta; \Gamma \vdash e' : t$.

Lemma 5.7 (Progress). If $\Delta; \cdot \vdash e : t$ then either e is a value or there exists an e' such that $e \longrightarrow e'$.

Theorem 5.8 (Type Soundness). If $\Delta; \cdot \vdash e : t$ then either e is a value or there exists an e' such that $e \longrightarrow e'$ and $\Delta; \cdot \vdash e' : t$.

5.2 Strong Normalization for $\lambda_{2C}^{(0),\times,+}$

The proof of strong normalization for λ_{2C} given in the previous section can be extended to show strong normalization for $\lambda_{2C}^{(0),\times,+}$.

Definition 5.9 (Neutral). The neutral terms for $\lambda_{2C}^{(0),\times,+}$ include those for λ_{2C} , and additionally:

$$\text{fst } e \downarrow \quad \text{snd } e \downarrow \quad \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \downarrow$$

The definition of the reducibility candidates remains unchanged.

We need to extend the definition of the set of partial proofs to include the new proof terms.

Definition 5.10. The set of normal neutral terms, NN , is defined by

$$\text{NN} = \{e \mid e \downarrow \text{ and } e \text{ is in normal form}\}$$

The set of partial proofs, PP , is defined by

$$\begin{aligned}
\text{PP}_0 &= \text{NN} \\
\text{PP}_{n+1} &= \text{NN} \\
&\cup \{ \text{trans}(e_p, e_s), \text{trans}(e_s, e_p), \text{funI}(e_p, e_s), \text{funI}(e_s, e_p), \text{subI}(e_p, e_s), \text{subI}(e_s, e_p), \\
&\quad \text{allI}(e_p), \text{prodI}(e_p, e_s), \text{prodI}(e_s, e_p), \text{sumI}(e_p, e_s), \text{sumI}(e_s, e_p) \mid e_p \in \text{PP}_n, e_s \in \text{SN} \} \\
\text{PP}_\infty &= \bigcup_{i \in \mathbb{N}} \text{PP}_i \\
\text{PP} &= \text{cand}(\text{PP}_\infty)
\end{aligned}$$

It is easy to see that each of the lemmas proved for the old definition of PP still holds for the new one.

Definition 5.11. For sets of terms A and C , we define the continuation set C_x^A by

$$e \in C_x^A \iff \text{for all } e' \in A, [e'/x]e \in C$$

Given sets of terms A and B , we define

$$\begin{aligned}
A \times B &= \{e \mid \text{fst } e \in A \text{ and } \text{snd } e \in B\} \\
A + B &= \{e \mid \text{for all } C \in \mathcal{RC}, \text{ variables } x \text{ and } y, e_1 \in C_x^A, e_2 \in C_y^B, \\
&\quad \text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C\}
\end{aligned}$$

Lemma 5.12. For all variables x, y with $x \neq y$, and $C \in \mathcal{RC}$, $y \in C_x^A$.

Proof. $[e/x]y = y$ and $y \in C$ since its normal and neutral. □

Lemma 5.13. If $[e'/x]e \in \text{SN}_n$ then $e \in \text{SN}_n$.

Proof. By induction on n , using Lemma 4.25. □

Lemma 5.14. If $A \neq \emptyset$ and $C \in \mathcal{RC}$ then C_x^A satisfies CR1 and CR2.

Proof.

(CR1) Suppose $e \in C_x^A$ and $e' \in A$.

Then $[e'/x]e \in C$ (by definition of C_x^A)

$[e'/x]e \in \text{SN}$ (by CR1 for C)

$e \in \text{SN}$ (by Lemma 5.13)

(CR2) Suppose $e \in C_x^A$, $e \longrightarrow e'$.

Then $[e_1/x]e \in C$.

Fix an arbitrary $e_1 \in A$, $[e_1/x]e \longrightarrow [e_1/x]e'$ (by Lemma 4.25)

$[e_1/x]e' \in C$ (by CR2 for C)

$e' \in C_x^A$

□

Lemma 5.15. If $\text{fst } e \in \text{SN}_{n_1}$ and $\text{snd } e \in \text{SN}_{n_2}$ then $e \in \text{SN}_{n_1+n_2}$

Proof. By induction on $n_1 + n_2$. Suppose $e \longrightarrow e'$ then $\text{fst } e \longrightarrow \text{fst } e'$ and $\text{snd } e \longrightarrow \text{snd } e'$, so there exist $n'_1 < n_1$ and $n'_2 < n_2$ such that $\text{fst } e' \in \text{SN}_{n'_1}$ and $\text{snd } e' \in \text{SN}_{n'_2}$ so by the IH we have $e' \in \text{SN}_{n'_1+n'_2}$. $n'_1 + n'_2 < n_1 + n_2$ so $e \in \text{SN}_{n_1+n_2}$. □

Lemma 5.16. If case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in \text{SN}_n$ then $e \in \text{SN}_n$

Proof. By induction on n . Suppose $e \longrightarrow e'$ then

case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \longrightarrow$ case e' of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$

so there exists $n' < n$ such that case e' of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in \text{SN}_{n'}$. By the IH we have $e' \in \text{SN}_{n'}$, so $e \in \text{SN}_n$. □

Lemma 5.17. Given:

- $A, B, C \in \mathcal{RC}$,
- $e_1 \in C_x^A$ and $e_2 \in C_y^B$,
- $e_1 \in \text{SN}_{n_1}$ and $e_2 \in \text{SN}_{n_2}$, and
- $e \downarrow$,
- for all e' , $e \longrightarrow e'$ implies $e' \in A + B$,

case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$

Proof. We will use CR3. case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \downarrow$, and we will show for all e' , case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \rightarrow e'$ implies $e' \in C$, by induction on $n_1 + n_2$. We distinguish cases over the last rule of the reduction.

case: red-case1. Impossible because $e \downarrow$.

case: red-case2. Impossible because $e \downarrow$.

case: red-caseC1. Then $e' = \text{case } e'' \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$ and $e \rightarrow e''$, so $e'' \in A + B$. Therefore case e'' of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$.

case: red-caseC2. Then $e' = \text{case } e' \text{ of } \{\text{inl}(x) \rightarrow e'_1, \text{inr}(y) \rightarrow e_2\}$ and $e_1 \rightarrow e'_1$, so there exists $n'_1 < n_1$ such that $e'_1 \in \text{SN}_{n'_1}$. case e' of $\{\text{inl}(x) \rightarrow e'_1, \text{inr}(y) \rightarrow e_2\} \downarrow$ and by the IH,

case e of $\{\text{inl}(x) \rightarrow e'_1, \text{inr}(y) \rightarrow e_2\} \in C$.

case: red-caseC3. Then $e' = \text{case } e' \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e'_2\}$ and $e_2 \rightarrow e'_2$, so there exists $n'_2 < n_2$ such that $e'_2 \in \text{SN}_{n'_2}$. case e of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e'_2\} \downarrow$ and by the IH

case e' of $\{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e'_2\} \in C$.

□

Lemma 5.18. Given $A, B \in \mathcal{RC}$,

1. $A \times B \in \mathcal{RC}$

Proof.

(CR1) Assume $e \in A \times B$

fst $e \in A$ and snd $e \in B$

(by definition of \times)

fst $e \in \text{SN}$ and snd $e \in \text{SN}$

(by CR1 for A and B)

$e \in \text{SN}$

(by Lemma 5.15)

- (CR2) Assume $e \in A \times B$ and $e \longrightarrow e'$
 $\text{fst } e \in A$ and $\text{snd } e \in B$ (by definition of \times)
 $\text{fst } e \longrightarrow \text{fst } e'$ and $\text{snd } e \longrightarrow \text{snd } e'$ (by red-fstC and red-sndC)
 $\text{fst } e' \in A$ and $\text{snd } e' \in B$ (by CR2 for A and B)
 $e' \in A \times B$
- (CR3) Fix a term e such that $e \downarrow$
 Assume for all e' , $e \longrightarrow e'$ implies $e' \in A \times B$.
 Suppose $\text{fst } e \longrightarrow e''$, then $e'' = \text{fst } e'$ and $e \longrightarrow e'$,
 then $e' \in A \times B$, so $\text{fst } e' \in A$ (by definition of \times)
 $\text{fst } e \downarrow$, so $\text{fst } e \in A$ (by CR3 for A)
 Suppose $\text{snd } e \longrightarrow e''$, then $e'' = \text{snd } e'$ and $e \longrightarrow e'$,
 then $e' \in A \times B$, so $\text{snd } e' \in A$ (by definition of \times)
 $\text{snd } e \downarrow$, so $\text{snd } e \in A$ (by CR3 for A)
 $e \in A \times B$

□

2. $A + B \in \mathcal{RC}$

- (CR1) Assume $e \in A + B$.
 $y \in \text{SN}_x^A$ and $y \in \text{SN}_x^B$ (by Lemma 5.12)
 $\text{case } e \text{ of } \{\text{inl}(x) \rightarrow y, \text{inr}(x) \rightarrow y\} \in \text{SN}$ (by definition of $+$)
 $e \in \text{SN}$ (by Lemma 5.16)
- (CR2) Assume $e \in A + B$ and $e \longrightarrow e'$.
 Fix arbitrary $C \in \mathcal{RC}$, variables x and y , $e_1 \in C_x^A$, $e_2 \in C_y^B$,
 $\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$ (by definition of $+$)
 $\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$
 $\longrightarrow \text{case } e' \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\}$ (by red-caseC1)
 $\text{case } e' \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$ (by CR2 for C)
 $e' \in A + B$ (by definition of $+$)

(CR3) Assume $e \downarrow$ and for all e' such that

$$e \longrightarrow e', e' \in A + B$$

Fix arbitrary $C \in \mathcal{RC}$, variables x and y , $e_1 \in C_x^A$, $e_2 \in C_y^B$,

$$e_1 \in \text{SN} \text{ and } e_2 \in \text{SN} \quad (\text{by Lemma 5.14})$$

$$\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C \quad (\text{by Lemma 5.17})$$

$$e \in A + B \quad (\text{by definition of } +)$$

Interpretations of types remain the same, with an additional cases for $()$, $t_1 \times t_2$ and $t_1 + t_2$. We repeat the full definition for convenience.

Definition 5.19. Given a valuation ξ , and static term t we define $\llbracket t \rrbracket_\xi \subseteq \text{Terms}$ as follows:

$$\begin{aligned} \llbracket \alpha \rrbracket_\xi &= \xi(\alpha) \\ \llbracket t_1 \rightarrow t_2 \rrbracket_\xi &= \llbracket t_1 \rrbracket_\xi \rightarrow \llbracket t_2 \rrbracket_\xi \\ \llbracket \forall \alpha : \sigma. t \rrbracket_\xi &= \forall_\sigma (\lambda (A \in \llbracket \sigma \rrbracket). \llbracket t \rrbracket_{\xi[\alpha \mapsto A]}) \\ &= \bigcap_{A \in \mathcal{RC}} \llbracket t \rrbracket_{\xi[\alpha \mapsto A]} \\ \llbracket (t_1 \dot{\leq} t_2) \rrbracket_\xi &= (\llbracket t_1 \rrbracket_\xi \dot{\leq} \llbracket t_2 \rrbracket_\xi) \\ \llbracket () \rrbracket_\xi &= \text{SN} \\ \llbracket t_1 \times t_2 \rrbracket_\xi &= \llbracket t_1 \rrbracket_\xi \times \llbracket t_2 \rrbracket_\xi \\ \llbracket t_1 + t_2 \rrbracket_\xi &= \llbracket t_1 \rrbracket_\xi + \llbracket t_2 \rrbracket_\xi \end{aligned}$$

Lemma 5.20 (Interpretation is Compositional). For all types t, t' and valuations ξ ,

$$\llbracket [t'/\alpha]t \rrbracket_\xi = \llbracket t \rrbracket_{\xi[\alpha \mapsto \llbracket t' \rrbracket_\xi]}$$

Proof. By induction on the structure of t . Each case but $t_1 \times t_2$ is the same as in the proof of Lemma 4.21

case: $t = ()$. Then $[t'/\alpha]t = t$.

case: $t = t_1 \times t_2$.

$$\begin{aligned}
[[t'/\alpha]t_1 \times t_2]_\xi &= [[t'/\alpha]t_1]_\xi \times [[t'/\alpha]t_2]_\xi \\
&\stackrel{IH}{=} [[t_1]_{\xi[\alpha \mapsto [t']_\xi]}] \times [[t_2]_{\xi[\alpha \mapsto [t']_\xi]}] \\
&= [[t_1 \times t_2]_{\xi[\alpha \mapsto [t']_\xi]}]
\end{aligned}$$

case: $t = t_1 + t_2$.

$$\begin{aligned}
[[t'/\alpha]t_1 + t_2]_\xi &= [[t'/\alpha]t_1]_\xi + [[t'/\alpha]t_2]_\xi \\
&\stackrel{IH}{=} [[t_1]_{\xi[\alpha \mapsto [t']_\xi]}] + [[t_2]_{\xi[\alpha \mapsto [t']_\xi]}] \\
&= [[t_1 + t_2]_{\xi[\alpha \mapsto [t']_\xi]}]
\end{aligned}$$

□

The following Lemma is used to show that the product introduction rule is sound.

Lemma 5.21. If $A, B \in \mathcal{RC}$, $e_1 \in A$ and $e_2 \in B$ then $\text{fst}(e_1, e_2) \in A$.

Proof. By CR1 for A and B , there exist n_1, n_2 with $e_1 \in \text{SN}_{n_1}$ and $e_2 \in \text{SN}_{n_2}$. We have $\text{fst}(e_1, e_2) \downarrow$. By induction on $n_1 + n_2$ we will show that if $\text{fst}(e_1, e_2) \longrightarrow e'$ then $e' \in A$, and use CR3 to get the desired conclusion.

Fix e' with $\text{fst}(e_1, e_2) \longrightarrow e'$

Case on the last rule on the derivation of the reduction

[red-fst] Then $e' = e_1 \in A$

[red-fstC] Then $e' = (e'_1, e'_2)$

The only reductions that apply to (e_1, e_2) are congruences,

so $e'_2 = e_2$ and $e_1 \longrightarrow e'_1$,

or $e'_1 = e_1$ and $e_2 \longrightarrow e'_2$

In either case the IH gives $(e'_1, e'_2) \in A \times B$.

□

Lemma 5.22. If $A, B \in \mathcal{RC}$, $e_1 \in A$ and $e_2 \in B$ then $\text{snd}(e_1, e_2) \in B$.

Proof. Similar to that for Lemma 5.21.

□

Lemma 5.23. Given

- $A, B \in \mathcal{RC}$
- $e_1 \in \text{SN}_{n_1}, e_2 \in \text{SN}_{n_2},$

if $e_1 \in A$, and $e_2 \in B$, then $(e_1, e_2) \in A \times B$.

Proof. Follows directly from Lemmas 5.21 and 5.22. □

The final lemmas are used to show that subtyping introduction and elimination for product types is sound.

Lemma 5.24. If $A \subseteq A'$ and $B \subseteq B'$ then $A \times B \subseteq A' \times B'$.

Proof.

Assume $e \in A \times B$.

$\text{fst } e \in A$ and $\text{snd } e \in B$

(by definition of \times)

$\text{fst } e \in A'$ and $\text{snd } e \in B'$

$e \in A' \times B'$

□

Lemma 5.25. If $A \times B \subseteq A' \times B'$ then $A \subseteq A'$ and $B \subseteq B'$

Proof.

Assume $e_1 \in A$ and $e_2 \in B$.

$\text{fst } (e_1, e_2) \in A$ and $\text{snd } (e_1, e_2) \in B$

(by Lemmas 5.21 and 5.22)

$(e_1, e_2) \in A \times B$

(by definition of \times)

$(e_1, e_2) \in A' \times B'$

$\text{fst } (e_1, e_2) \in A'$ and $\text{snd } (e_1, e_2) \in B'$

$\text{fst } (e_1, e_2) \longrightarrow e_1$ and $\text{snd } (e_1, e_2) \longrightarrow e_2$

$e_1 \in A'$ and $e_2 \in B'$

(by CR2 for A' and B')

□

Lemma 5.26. If for all $e' \in A$, $\llbracket e \rrbracket_{\eta[x \mapsto e']} \in C$, $\llbracket e \rrbracket_{\eta[x \mapsto x]} \in C_x^A$.

Proof. By induction on the structure of e . □

Lemma 5.27. If $A \subseteq B$ then $C_x^B \subseteq C_x^A$.

Proof.

Assume $e \in C_x^B$.

Then for all $e' \in B$, $[e'/x]e \in C$.

Fix $e'' \in A$,

$$e'' \in B$$

(by $A \subseteq B$)

$$[e''/x]e \in C$$

$$e \in C_x^A.$$

□

Lemma 5.28. If $A \subseteq A'$ and $B \subseteq B'$ then $A + B \subseteq A' + B'$.

Proof.

Assume $e \in A + B$.

For all $C \in \mathcal{RC}$, variables x and y , $e_1 \in C_x^A$ and $e_2 \in C_y^B$

$$\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$$

Fix arbitrary $C \in \mathcal{RC}$, variables x and y , $e_1 \in C_x^{A'}$ and $e_2 \in C_y^{B'}$

$$C_x^{A'} \subseteq C_x^A, \text{ and } C_x^{B'} \subseteq C_x^B$$

(by Lemma 5.27)

$$e_1 \in C_x^A \text{ and } e_2 \in C_y^B$$

$$\text{case } e \text{ of } \{\text{inl}(x) \rightarrow e_1, \text{inr}(y) \rightarrow e_2\} \in C$$

$$e \in A' + B'$$

□

Lemma 5.29. If $A + B \subseteq A' + B'$ then $A \subseteq A'$ and $B \subseteq B'$

Proof.

Assume $e_1 \in A$ and $e_2 \in B$.

Then $\text{inl}(e_1) \in A' + B'$ and $\text{inr}(e_2) \in A' + B'$.

So case $\text{inl}(e_1)$ of $\{\text{inl}(x) \rightarrow x, \text{inr}(x) \rightarrow y\} \in A'$, and

$$\text{case } \text{inl}(e_2) \text{ of } \{\text{inl}(x) \rightarrow y, \text{inr}(x) \rightarrow x\} \in B'$$

(by definition of +)

case $\text{inl}(e_1)$ of $\{\text{inl}(x) \rightarrow x, \text{inr}(x) \rightarrow y\} \longrightarrow e_1$, and

$$\text{case } \text{inl}(e_2) \text{ of } \{\text{inl}(x) \rightarrow y, \text{inr}(x) \rightarrow x\} \longrightarrow e_2$$

$$e_1 \in A' \text{ and } e_2 \in B'$$

(by CR2 for A' and B')

□

Using these facts it is straightforward to prove strong normalization for $\lambda_{2C^-}^{(), \times, +}$.

Theorem 5.30 (Strong Normalization). If $\Delta; \cdot \vdash e : t$ then $e \in \text{SN}$.

6 Related Work

There have been a number of systems that make use of constraints on type variables in order to encode GADTs. Xi et. al. describe guarded recursive datatype constructors (GRDCs) [17], which are recursive datatypes that include constraints on type parameters. The constraints used by GRDCs are different from constraint types in that they are not themselves types, but instead are part of the type system. With GRDCs, there are no witnesses for constraints, but instead constraint assertions and assumptions are embedded into types and a decision procedure is used to solve for constraint satisfaction. Similarly, Simonet and Pottier describe a constraint-based formulation of GADTs [12] in which constraints are attached to type quantifiers and a decision procedure is used to solve constraints. Along similar lines, ATS [16] is a language which includes GRDCs along with Dependent ML-style [15] type indexes and index constraints. ATS also lacks witnesses for type constraint satisfaction and uses a decision procedure.

System F with type equality coercions [13] (System F_C), extends System F with sorts for type-equality constraints. Like System F with constraint types, System F_C includes type constraints and witnesses for type constraint satisfaction. However, in System F_C constraint satisfaction witnesses exist at the type level, and constraints are expressed by sorts. The formulation of System F_C is similar to that of System F with constraint types, however System F_C has not been proven to be strongly normalizing.

7 Conclusion

We have described several calculi which extend System F with types that assert subtyping relationships between other types. Members of these subtyping constraint types are witnesses that the constraint holds. A significant advantage of having explicit witnesses is that terms can be easily type-checked without resorting to the use of a complicated decision procedure.

We have proved type soundness for λ_{2C} , a powerful formulation of System F with constraint types. This shows that constraint types may be added to polymorphic programming languages, even those with impredicative quantification, without breaking type soundness.

We have proved strong normalization for λ_{2C^-} and $\lambda_{2C^-}^{(0,\times,+)}$, two restrictions of λ_{2C} which lack some subtyping elimination rules. This shows that System F with constraint types corresponds to a sound logic, and may form the basis of a theorem proving language. The strong normalization proofs given are relatively simple extensions Girard's original reducibility candidates argument. This stands in contrast to the more complicated proofs given for other theorem proving languages like the Calculus of Constructions.

We have also found that the addition of constraint types alone to System F is not enough to allow for encoding of GADTs. We have shown that the addition of type-level functions and higher-order polymorphism can remedy this limitation.

References

- [1] Homepage for ATS. <http://www.cs.bu.edu/~hwxi/ATS/ATS.html>.
- [2] The Coq proof assistant. <http://coq.inria.fr/>.
- [3] Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL '85: Invited Lectures from the European Conference on Computer Algebra-Volume I*, pages 151–184, London, UK, 1985. Springer-Verlag.
- [4] The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>.
- [5] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris, 1972.
- [6] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.
- [7] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, 1999.
- [8] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, New York, NY, USA, 1995. ACM.
- [9] Haskell 98 language report. <http://www.haskell.org/onlinereport/>.
- [10] The Caml language: Home. <http://caml.inria.fr/>.
- [11] Robert Harper Robin Milner, Mads Tofte and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [12] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1):1, 2007.
- [13] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

- [14] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, June 1967.
- [15] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, Boston, June 2001.
- [16] Hongwei Xi. Applied type system (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.
- [17] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.