Chapter 2 (draft): a Dependent Type System

Mark Lemay

December 10, 2021

Part I

Surface language

Despite the usability issues this thesis hopes to correct, dependent type systems are still one of the most promising technologies for correctness in programming. Since proofs and programs are associated there is no additional syntax for programmers to learn. The proof systems is predictable from the perspective of a functional programmer.

The **surface language** presented in this chapter specifies a minimal dependent type system. The semantics are intended to be as simple as possible and compatible with other well studied intentional dependent type theories. It has several (but not all) of the standard properties of dependent type theory. As much as possible, the syntax uses standard modern notation [1].

should cit

Martin Lo
pr notatio

The surface language will serve both as foundation for later chapters and a self contained technical introduction to dependent types. By design, the machinery of later chapters should be invisible to programers that use the full system. They should only need to think about the surface language. Everything presented in later chapters is designed to reinforce an understanding of the surface type system, and make it easier to use.

The surface language deviates from a standard dependent type theory to include features to ease programming at the expense of logical correctness. Specifically the language allows general recursion, since general recursion is useful for general purpose functional programming. Type-in-type is also supported since it simplifies the system for programmers, and makes the meta-theory slightly easier. Despite this, type soundness is a achievable and a practical type checking system is given.

Though similar systems have been studied over the last few decades this chapter aims to give a self contained presentation, along with examples. The surface language has been an good platform to conduct research into full spectrum dependent type theory, and hopefully this exposition will be helpful introduction for other researchers.

is there a
then surfa

1 Surface Language Syntax

The syntax for the surface language is in figure 1. The syntax supports: variables, type annotations, a single type universe, dependent function types, recursive dependent functions, and function applications. Type annotations are written with two colons to differentiate it from the formal typing judgments that will appear more frequently in this text, in the implemented language a user of the programming language would use a single colon.

There is no destination between types and terms in the syntax[2], both are referred to as expressions. However, capital metavariables are used in positions that are intended as types, and lowercase metavariables are used when an expression is intended to be a term, for instance in annotation syntax.

more abou
sive?

Several standard abbreviations are listed in 2 for convenience.

[1] several alternative syntaxes have existed in the literature. In this document the typed polymorphic identity function is written, $\lambda - x \Rightarrow x \ : (X : \star) \to X \to X$. In [CH88] it might be written $(\lambda X : \star)\,(\lambda x : X)\, x \ : [X : \star]\,[x : X]\, X$. In [Pro13] it might be written $\lambda X.\lambda x.x \ : \prod_{(X : \mathcal{U})} X \to X$.

[2] terms and types are usually separated, except in the syntax of full-spectrum dependent type systems where separating them would require many redundant rules

variable identifier,
x, y, z, f
type contexts,

$$\Gamma \quad ::= \quad \Diamond \mid \Gamma, x : M$$

expressions,

$$
\begin{array}{llll}
m, n, M, N & ::= & x & \text{variable} \\
& \mid & m :: M & \text{annotation} \\
& \mid & \star & \text{type universe} \\
& \mid & (x : M) \to N & \text{function type} \\
& \mid & \mathsf{fun}\, f\, x \Rightarrow m & \text{function} \\
& \mid & m\, n & \text{application}
\end{array}
$$

Figure 1: Surface Language Syntax

$$
\begin{array}{lllll}
(x : M) \to N & \text{written} & M \to N & \text{when} & x \notin fv\,(N) \\
\mathsf{fun}\, f\, x \Rightarrow m & \text{written} & \lambda x \Rightarrow m & \text{when} & f \notin fv\,(m) \\
...\, x \Rightarrow \lambda y \Rightarrow m & \text{written} & ...\, x\, y \Rightarrow m & & \\
x & \text{written} & - & \text{when} & x \notin fv\,(m) \text{ when } x \text{ binds } m
\end{array}
$$

where fv is a function that returns the set of free variables in an expression

Figure 2: Surface Language Abbreviations

2 Examples

The surface system is extremely expressive. Several example surface language constructions can be found in 2. Turnstile notion is abused slightly so that examples can be indexed by other expressions that obey type rules. For instance, we can say $refl_{2_c:\mathbb{N}_c}\ :\ 2_c \doteq_{\mathbb{N}_c} 2_c$ since $\mathbb{N}_c : \star$ and $2_c : \mathbb{N}_c$.

2.1 Church encodings

Data types are expressible using Church encodings, (in the style of System F). Church encodings embed the elimination principle of a data type into continuationsFor instance Boolean data is eliminated against true and false, 2 tags with no additional data. This can also be recognized as the familiar if-then-else construct. So $\mathbb{B}_c$ encodes the possibility of choice between 2 elements, $true_c$ picks the “then” branch, and $false_c$ picks the “else” branch.

Natural numbers[3] are encodable with two tags, 0 and successor. Where successor also contains the result of the preceding number. So $\mathbb{N}_c$ encodes those two choices, $(X \to X)$ handles the recursive result of the prior number in the successor case, and the X argument specifies how to handle the base case of 0. This can be viewed as a simple looping construct with temporary storage.

Parameterized data types such as pairs and the *Either* type can also be encoded in this scheme. A pair type can be used in any way the two terms it contains can, so the definition states that a pair is at least as good as the curried input to a function. The *Either* type is handled if both possibilities are handled, which is expressed by it’s definition.

Church encodings provide a theoretically light weight way of working with data in minimal lambda calculus, however they are very inconvenient to work with. For instance, the predecessor function on natural numbers is not straight forward. To make the system easier for programmers, data types will be addressed in Chapter 4.

2.2 Proposition encodings

In general we associate the truth value of a proposition with the inhabitation of a type by a meaningful value. So, $\perp_c$, the “empty” type, can be considered as a false proposition. While $Unit_c$ can be considered a trivially true logical proposition.

Several of the church encoded data types we have seen can also be interpreted as logical predicates. For instance, the tuple type can be considered as logical and, $X \times_c Y$ can be inhabited exactly when both X and Y are inhabited. The *Either* type can be considered as logical or, $Either_c\, X\, Y$ can be inhabited exactly when either X or Y is inhabited.

[3]called “church numerals”

	$\vdash \bot_c$	$:= (X : \star) \to X$	$: \star$	Void, “empty” t
	$\vdash Unit_c$	$:= (X : \star) \to X \to X$	$: \star$	Unit, logical tr
	$\vdash tt_c$	$:= \lambda -\ x \Rightarrow x$	$: Unit_c$	trivial proposit
	$\vdash \mathbb{B}_c$	$:= (X : \star) \to X \to X \to X$		booleans
	$\vdash true_c$	$:= \lambda -\ then - \Rightarrow then$	$: \mathbb{B}_c$	boolean true
	$\vdash false_c$	$:= \lambda - -\ else \Rightarrow else$	$: \mathbb{B}_c$	boolean false
$x : \mathbb{B}_c$	$\vdash !_c x$	$:= x\, \mathbb{B}_c\, false_c\, true_c$	$: \mathbb{B}_c$	boolean not
$x : \mathbb{B}_c, y : \mathbb{B}_c$	$\vdash x \,\&_c\, y$	$:= x\, \mathbb{B}_c\, y\, false_c$	$: \mathbb{B}_c$	boolean and
	$\vdash \mathbb{N}_c$	$:= (X : \star) \to (X \to X) \to X \to X$	$: \star$	natural number
	$\vdash 0_c$	$:= \lambda - -\ z \Rightarrow z$	$: \mathbb{N}_c$	
	$\vdash 1_c$	$:= \lambda -\ s\, z \Rightarrow s\, z$	$: \mathbb{N}_c$	
	$\vdash 2_c$	$:= \lambda -\ s\, z \Rightarrow s\, (s\, z)$	$: \mathbb{N}_c$	
	$\vdash n_c$	$:= \lambda -\ s\, z \Rightarrow s^n\, z$	$: \mathbb{N}_c$	
$x : \mathbb{N}_c, y : \mathbb{N}_c$	$\vdash x +_c y$	$:= \lambda X\, s\, z \Rightarrow x\, X\, s\ (y\, X\, s\, z)$	$: \mathbb{N}_c$	
$X : \star, Y : \star$	$\vdash X \times_c Y$	$:= (Z : \star) \to (X \to Y \to Z) \to Z$	$: \star$	pair, logical an
$X : \star, Y : \star$	$\vdash Either_c\, X\, Y$	$:= (Z : \star) \to (X \to Z) \to (Y \to Z) \to Z$	$: \star$	either, logical o
$X : \star$	$\vdash \neg_c X$	$:= X \to \bot_c$	$: \star$	logical negation
$x : \mathbb{N}_c$	$\vdash Even_c\, x$	$:= \mathbb{N}_c\ \star\ (\lambda x \Rightarrow \neg_c x)\ Unit_c$	$: \star$	x is an even nu
$X : \star, Y : X \to \star$	$\vdash \exists_c x : X. Y\, x$	$:= (C : \star) \to ((x : X) \to Y\, x \to C) \to C$	$: \star$	dependent pair
$X : \star, x_1 : X, x_2 : X$	$\vdash x_1 \doteq_X x_2$	$:= (C : (X \to \star)) \to C\, x_1 \to C\, x_2$	$: \star$	Leibniz equalit
$X : \star, x : X$	$\vdash refl_{x:X}$	$:= \lambda -\ cx \Rightarrow cx$	$: x \doteq_X x$	reflexivity
$X : \star, x_1 : X, x_2 : X$	$\vdash sym_{x_1, x_2 : X}$	$:= \lambda p\, C \Rightarrow p\, (\lambda x \Rightarrow C\, x \to C\, x_1)\ (\lambda x \Rightarrow x)$	$: x_1 \doteq_X x_2 \to x_2 \doteq_X x_1$	symmetry

turn . to => in exists?

With dependent types, more interesting logical predicates can be encoded. For instance, we can characterize when a number is Even with $Even_c\, x$. We can show that 2 is even by showing that $Even_c\, 2_c$ is inhabited with with the term $\lambda s \Rightarrow s\, tt_c$.

Other predicates are encodable in the style of Calculus of Constructions[CH88] . For instance, we can encode the existential as $\exists_c$, then if we want to show $\exists_c x : \mathbb{N}_c. \, Even_c\, x$ we need to find a suitable inhabitant of that type. 0is clearly an even number, so our inhabitant could be $\lambda f \Rightarrow f\, 0_c\, tt_c$. Note that the existential degenerates into the tuple if Y does not depend on the first element.

One of the most potent and interesting propositions is the proposition of equality. $\doteq$ is referred to as Leibniz equality since two terms are equal when they behave the same on all “observations”[4]. We can prove $\doteq$ is an equivalence within the system be proving it is reflexive, symmetric, and transitive. Additionally we can prove congruence.

2.3 Large Eliminations

It is useful for a type to depend specifically on term level data, this is called **large elimination**. Large elimination can be simulated with type-in-type.

$toLogic$	$:= \lambda b \Rightarrow b\ \star\ Unit_c\ \bot_c$	:	$\mathbb{B}_c \to \star$
$isPos$	$:= \lambda n \Rightarrow n\ \star\ (\lambda - \Rightarrow Unit_c)\ \bot_c$	:	$\mathbb{N}_c \to \star$

For instance, $toLogic$ can convert a $\mathbb{B}_c$ term into its corresponding logical type, $toLogic\, true_c \equiv Unit_c$ while $toLogic\, false_c \equiv \bot_c$. The expression $isPos$ has similar behavior, going to $\bot_c$ at 0_c and $Unit_c$ otherwise.

Note that such functions are not possible in the Calculus of Constructions.

2.3.1 Inequalities

Large eliminations can be used to prove inequalities that can be hard or impossible to express in other minimal dependent type theories such as the Calculus of Constructions. For instance,

$\lambda pr \Rightarrow pr\ (\lambda x \Rightarrow x)\ \bot_c$	:	$\neg_c \star \doteq_\star \bot_c$	the type universe is distinct from Logical False
$\lambda pr \Rightarrow pr\ (\lambda x \Rightarrow x)\ tt_c$	:	$\neg_c Unit_c \doteq_\star \bot_c$	Logical True is distinct from Logical False
$\lambda pr \Rightarrow pr\, toLogic\, tt_c$	:	$\neg true_c \doteq_{\mathbb{B}_c} false_c$	boolean true and false are distinct
$\lambda pr \Rightarrow pr\, isPos\, tt_c$	:	$\neg 1_c \doteq_{\mathbb{N}_c} 0_c$	1 and 0 are distinct

[4]The identification of indiscernibles is called “Leibniz law” in philosophy. Leibniz assumed a metaphysical notion of identification of “substance”s, not a mathematical notion of equality.

$$\frac{x : M \in \Gamma}{\Gamma \vdash x\,:\,M}\ \text{ty-var}$$

$$\frac{\Gamma \vdash m\,:\,M}{\Gamma \vdash m :: M\,:\,M}\ \text{ty-::}$$

$$\frac{}{\Gamma \vdash \star\,:\,\star}\ \text{ty-}\star$$

$$\frac{\Gamma \vdash M\,:\,\star \quad \Gamma, x : M \vdash N\,:\,\star}{\Gamma \vdash (x : M) \rightarrow N\,:\,\star}\ \text{ty-fun-ty}$$

$$\frac{\Gamma \vdash m\,:\,(x : N) \rightarrow M \quad \Gamma \vdash n\,:\,N}{\Gamma \vdash m\,n\,:\,M\,[x \coloneqq n]}\ \text{ty-fun-app}$$

$$\frac{\Gamma, f : (x : N) \rightarrow M, x : N \vdash m\,:\,M}{\Gamma \vdash \mathsf{fun}\,f\,x \Rightarrow m\,:\,(x : N) \rightarrow M}\ \text{ty-fun}$$

$$\frac{\Gamma \vdash m\,:\,M \quad M \equiv M'}{\Gamma \vdash m\,:\,M'}\ \text{ty-conv}$$

font stuff

Figure 3: Surface Language Type Assignment System

Note that a proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ is not possible in the Calculus of Constructions[Smi88][5].

2.4 Recursion

Additionally, the syntax of functions builds in unrestricted recursion. Though not always necessary, recursion can be very helpful for writing programs. For instance, here is (an inefficient) function that calculates Fibonacci numbers.

$\mathsf{fun}\,f\,x \Rightarrow case_c\,x\,0_c\,(\lambda px \Rightarrow case_c\,px\,1_c\,(\lambda - \Rightarrow f\,(x -_c 1) +_c f\,(x -_c 2)))$

Assuming appropriate definitions for $case_c$, and subtraction.

Recursion can also be used to simulate induction, and this will be heavily relied on when data types are added in chapter 4.

3 Surface Language Type Assignment System

When is an expression reasonable? The expression $\star\star\star\star$ is allowed by the grammar of the language, but seems dubiousType systems can disallow "bad terms" like these which in turn avoids bad runtime behavior.

We will present our our type system as a type assignment system. Practically this means that the type assignment system may need to infer an impractical amount of information from a term and its context for typing. This also means that terms do not necessarily have unique typings. For instance $\vdash \lambda x \Rightarrow x : \mathbb{N}_c \rightarrow \mathbb{N}_c$, and $\vdash \lambda x \Rightarrow x : \mathbb{B}_c \rightarrow \mathbb{B}_c$.

The rules of the type assignment system are listed in 3[6]. Variables get their type from the typing context. Type annotations reflect a correct typing derivation in the ty-:: rule. Type-in-type is recognized by the ty-$\star$ rule. The ty-fun-ty rule forms dependent function types. The ty-fun-app rule shows how to type function application, by substituting the argument term directly into the dependent function type. Functions are typed with a variable for recursive calls along with a variable for their argument ty-fun. Finally, ty-conv shows which types are **convertible** to each other.

the most important property of a type system is **type soundness**[7]. Type soundness is often motivated with the slogan, "well typed programs don't get stuck"[Mil78][8]. Given the syntax of the surface language, their is potential

[5] Martin Hofmann excellently motivates the reasoning in [Hof97b]Exercises 2.5, 2.6, 3.7, 3.25, 3.26, 3.43, 3.44

[6] There is some question about how much typing information should be coupled to the judgment, forcing contexts to be well formed eliminates nonsense situation like $x : 1_c \vdash ...$ by construction, but requires more fork when forming judgments that can be distracting. The proofs in this section can be done without forcing the context to be well formed, the additional constraints are omitted.

[7] also called "type safety"

[8] in Milner's original paper, he used "wrong" instead of stuck

for a program to "get stuck" when an argument is applied to a non-function constructor. For example, $\star\ 1_c$ would be stuck since $\star$ is not a function, so it cannot compute when given the argument 1_c. A good type system will make such unreasonable programs impossible.

Type soundness can be shown with a progress and preservation[9] style proof[WF94][10]

The preservation lemma shows that typing information is invariant over evaluation. While the progress lemma shows that a single step of evaluation for a well typed term in an empty context will not "get stuck". By iterating these lemmas together, it is possible to show that the type system prevents a term from evaluating to the class of bad behavior described above. This type of proof hinges on a suitable definition of the $\equiv$ relation.

The $\equiv$ relation characterizes when terms are "obviously" equal, or "automatically" equal. Because the $\equiv$ relation is usually based on the definition of terms, rather then on extrinsic properties, it is called **definitional equality**. Usually it is desirable to make the definitional equality relation as large as possible, though we will not do that in this chapter. Chapter 3 will pose an alternative way to avoid definitional equalities.

is that ac

In a progress and preservation style proof the $\equiv$ relation should

- be reflexive, $m \equiv m$
- be symmetric, if $m \equiv m'$ then $m' \equiv m$
- be transitive, if $m \equiv m'$ and $m' \equiv m''$ then $m \equiv m''$
- be closed under substitutions and evaluation, for instance if $m \equiv m'$ and $n \equiv n'$ then $m\,[x \coloneqq n] \equiv m'\,[x \coloneqq n']$
- distinguish between type constructors, for instance $\star \not\equiv (x : N) \to M$

A particularly simple definition of $\equiv$ is equating any terms that share a reduct via a system of parallel reductions

$$\frac{m \Rrightarrow_* n \quad m' \Rrightarrow_* n}{m \equiv m'} \equiv\text{-Def}$$

this relation

- is reflexive, by definition
- is symmetric, automatically
- is transitive, if $\Rrightarrow_*$ is confluent
- is closed under substitution if $\Rrightarrow_*$ is closed under substitution, closed under evaluation automatically
- distinguishes type constructors, if they are stable under reduction. For instance, if
  - $\forall NM.\, (x : N) \to M \Rrightarrow P$ implies $P = (x : N') \to M'$
  - and $\star \Rrightarrow P$ implies $P = \star$
  - then $(x : N) \to M \not\equiv \star$

Parallel reductions are defined to make confluence easy to prove, by allowing the simultaneous evaluation of any available reduction. The system of parallel reductions is defined in 4 The only interesting rules are $\Rrightarrow$-fun-app-red and $\Rrightarrow$-::-red since they directly preform reductions. The other rules are entirely structural. Repeating a parallel reductions zero or more times is written $\Rrightarrow_*$.

While this is a sufficient presentation of definitional equality, others variants of the relation are possible. For instance it is possible to extend the relation with contextual information, type information, explicit proofs of equality (as in Extensional Type Theory), uncomputable relations (as in [JZSW10]). It is also common to assume the properties of $\equiv$ hold without proof.

3.1 Definitional Equality

We now have enough information to prove the critical properties of definitional equality.

[9] also called "Subject Reduction"

[10] Though first published in [WF94] their progress lemma is a bit different form modern presentations. Most relevant textbooks outline forms of this proof for non-dependent type systems. For instance, part 2 of [Pie02], [KSW20], the Chapala book

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{(\mathsf{fun}\, f\, x \Rightarrow m)\, n \Rrightarrow m' \,[f := \mathsf{fun}\, f\, x \Rightarrow m', x := n']} \Rrightarrow\text{-fun-app-red}$$

$$\frac{m \Rrightarrow m'}{m :: M \Rrightarrow m'} \Rrightarrow\text{-::-red}$$

$$\frac{}{x \Rrightarrow x} \Rrightarrow\text{-var}$$

$$\frac{m \Rrightarrow m' \quad M \Rrightarrow M'}{m :: M \Rrightarrow m' :: M'} \Rrightarrow\text{-::}$$

$$\frac{}{\star \Rrightarrow \star} \Rrightarrow\text{-}\star$$

$$\frac{M \Rrightarrow M' \quad N \Rrightarrow N'}{(x : M) \to N \Rrightarrow (x : M') \to N'} \Rrightarrow\text{-fun-ty}$$

$$\frac{m \Rrightarrow m'}{\mathsf{fun}\, f\, x \Rightarrow m \;\Rrightarrow\; \mathsf{fun}\, f\, x \Rightarrow m'} \Rrightarrow\text{-fun}$$

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{m\, n \Rrightarrow m'\, n'} \Rrightarrow\text{-fun-app}$$

$$\frac{}{m \Rrightarrow_* m} \Rrightarrow_*\text{-refl}$$

$$\frac{m \Rrightarrow_* m' \quad m' \Rrightarrow m''}{m \Rrightarrow_* m''} \Rrightarrow_*\text{-trans}$$

Figure 4: Surface Language Parallel Reductions

3.1.1 $\Rrightarrow$, $\Rrightarrow_*$, $\equiv$ are reflexive

The following rule is admissible,

$$\frac{}{m \Rrightarrow m} \Rrightarrow\text{-refl}$$

by induction on the syntax of m

Recall that $\Rrightarrow_*$ is reflexive by definition so

$$\frac{}{m \equiv m} \equiv\text{-refl}$$

is admissible.

3.1.2 $\Rrightarrow$, $\Rrightarrow_*$, $\equiv$ are closed under substitutions.

ons, what
eeded
ns?

eded or
mmodate
tuff in

The following rule is admissible for every substitution σ

$$\frac{m \Rrightarrow m'}{m\,[\sigma] \Rrightarrow m'\,[\sigma]} \Rrightarrow\text{-sub-}\sigma$$

by induction on the $\Rrightarrow$ relation, using $\Rrightarrow$-refl in the $\Rrightarrow$-var case.

The following rule is admissible where σ, τ is a substitution where for every x, $\sigma\,(x) \Rrightarrow \tau\,(x)$, written $\sigma \Rrightarrow \tau$

$$\frac{m \Rrightarrow m' \quad \sigma \Rrightarrow \tau}{m\,[\sigma] \Rrightarrow m'\,[\tau]} \Rrightarrow\text{-sub}$$

by induction on the $\Rrightarrow$ relation.

$$\frac{m \Rrightarrow_* m' \quad \sigma \Rrightarrow \tau}{m\,[\sigma] \Rrightarrow_* m'\,[\tau]} \Rrightarrow_*\text{-sub}$$

is admissible by induction on the $\Rrightarrow_*$ relation. And follows that

$$\frac{m \equiv m' \quad \sigma \Rrightarrow \tau}{m\,[\sigma] \equiv m'\,[\tau]}\ \equiv\text{-sub}$$

is admissible.

The following corollary is admissible

$$\frac{n \Rrightarrow_* n'}{m\,[x \coloneqq n] \equiv m\,[x \coloneqq n']}$$

since

$m \Rrightarrow_* m$	$\Rrightarrow_*$-refl
$m\,[x \coloneqq n] \Rrightarrow_* m\,[x \coloneqq n']$	by repeated $\Rrightarrow_*$-sub
$m\,[x \coloneqq n'] \Rrightarrow_* m\,[x \coloneqq n']$	$\Rrightarrow_*$-refl
$m\,[x \coloneqq n] \equiv m\,[x \coloneqq n']$	$\equiv$ -Def

3.1.3 $\Rrightarrow$, $\Rrightarrow_*$ is confluent, $\equiv$ is transitive

A relation R is **confluent**[11] when, for all m, n, n', ifmRn and mRn' then their exists n'' such that nRn'' and $n'Rn''$. If a relation is confluent, in a sense, specific paths don't matter since you can alway reach the same destinations.

Since we defined our normalization by parallel reductions we can show confluence following the proof in [Tak95][12]. First, define a function max that takes the maximum possible parallel step, such that if $m \Rrightarrow m'$ then $m' \Rrightarrow max\,(m)$ and $m \Rrightarrow max\,(m)$. This is referred to as the triangle property (a diagram is presented in 5).

$max($	$(\mathsf{fun}\,f\,x \Rightarrow m)\ n$	$) =$	$max\,(m)\,[f \coloneqq \mathsf{fun}\,f\,x \Rightarrow max\,(m)\,, x \coloneqq max\,(n)]$	otherwise
$max($	x	$) =$	x	
$max($	$m :: M$	$) =$	$max\,(m)$	
$max($	$\star$	$) =$	$\star$	
$max($	$(x : M) \to N$	$) =$	$(x : max\,(M)) \to max\,(N)$	
$max($	$\mathsf{fun}\,f\,x \Rightarrow m$	$) =$	$\mathsf{fun}\,f\,x \Rightarrow max\,(m)$	
$max($	$m\,n$	$) =$	$max\,(m)\ max\,(n)$	

If $m \Rrightarrow m'$ then $m' \Rrightarrow max\,(m)$.

by induction on the derivation $m \Rrightarrow m'$, with the only interesting cases are where a reduction is not taken

- in the case of $\Rrightarrow$-:: , $m' \Rrightarrow max\,(m)$ by $\Rrightarrow$-::-red
- in the case of $\Rrightarrow$-fun-app , $m' \Rrightarrow max\,(m)$ by $\Rrightarrow$-fun-app-red

It follows that, if $m \Rrightarrow m'$, $m \Rrightarrow m''$, implies $m' \Rrightarrow max\,(m)$,$m'' \Rrightarrow max\,(m)$ (referred to as the diamond property). Since the $max\,(m) = max\,(m)$.

The diamond property implies the confluence of $\Rrightarrow_*$, by repeated application of the diamond property.

It follows that $\equiv$ is transitive. Since if $m \equiv m'$ and $m' \equiv m''$ then by definition for some n, n', $m \Rrightarrow_* n$, $m' \Rrightarrow_* n$ and $m' \Rrightarrow_* n'$, $m'' \Rrightarrow_* n'$. If $m' \Rrightarrow_* n$ and $m' \Rrightarrow_* n'$. Then by confluence there exists some $p$such that $n \Rrightarrow_* p$ and $n' \Rrightarrow_* p$. By transitivity $m \Rrightarrow_* p$ and $m'' \Rrightarrow_* p$. So by definition $m \equiv m''$.

3.1.4 Stability

$\forall N, M, P.\, (x : N) \to M \Rrightarrow_* P \ \Rightarrow\ \exists N', M'.P = (x : N') \to M' \wedge N \Rrightarrow_* N' \wedge M \Rrightarrow_* M'$

by induction on $\Rrightarrow_*$

$\Rrightarrow_*$-refl	$P = (x : N) \to M$	
	$N \Rrightarrow_* N$	$\Rrightarrow_*$-refl
	$M \Rrightarrow_* M$	$\Rrightarrow_*$-refl
$\Rrightarrow_*$-trans	$(x : N) \to M \Rrightarrow_* P'$, $P' \Rrightarrow P''$	
	$P' = (x : N') \to M'$, $N \Rrightarrow_* N'$, $M \Rrightarrow_* M'$	by induction with $(x : N) \to M \Rrightarrow_* P'$
	$P'' = (x : N'') \to M''$, $N' \Rrightarrow N''$, $M' \Rrightarrow M''$	by inspection, only the $\Rrightarrow$-fun-ty rule is possible
	$N \Rrightarrow_* N''$	$\Rrightarrow_*$-trans
	$M \Rrightarrow_* M''$	$\Rrightarrow_*$-trans

Therefore the following rule is admissible

[11] also called "Church-Rosser"

[12] also well presented in [KSW20]

Triangle Property
$\forall m, m'.\, m \Rrightarrow m' \;\to\; m' \Rrightarrow max\,(m)$

$$\begin{tikzcd} m \arrow[r, Rrightarrow] \arrow[d] & m' \arrow[dl, Rrightarrow] \\ max(m) & \end{tikzcd}$$

Diamond Property
$\forall m, m', m''.\, m \Rrightarrow m' \;\wedge\; m \Rrightarrow m'' \;\to\; m' \Rrightarrow max\,(m)$

$$\begin{tikzcd} & m \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd] & \\ m' \arrow[dr, Rrightarrow] & & m'' \arrow[dl, Rrightarrow] \\ & max(m) & \end{tikzcd}$$

Confluence
$\forall m, n, n'.\, m \Rrightarrow_* n \;\wedge\; m \Rrightarrow_* n' \;\to\; \exists n'''.\, n \Rrightarrow_* n''' \;\wedge\; n' \Rrightarrow n'''$

$$\begin{tikzcd} & m \arrow[dl, Rrightarrow, "*"] \arrow[dr, Rrightarrow, "*"] & \\ n' \arrow[dr, Rrightarrow, "*"] & & n'' \arrow[dl, Rrightarrow, "*"] \\ & n''' & \end{tikzcd}$$

Figure 5: Rewriting Diagrams

$$\frac{(x : N) \to M \equiv (x : N') \to M'}{N \equiv N' \quad M \equiv M'}$$

$$\begin{array}{ll}
(x : N) \to M \Rrightarrow_* P, \quad (x : N') \to M' \Rrightarrow_* P & \text{by expending the definition of } \equiv \\
P = (x : N'') \to M'',\ N \Rrightarrow_* N'',\ M \Rrightarrow_* M'',\ N' \Rrightarrow_* N'',\ M' \Rrightarrow_* M'' & \text{by the lemma above} \\
N \equiv N' & \text{by the definition of } \equiv \text{ with } N \Rrightarrow_* N'',\ N' \Rrightarrow_* N'' \\
M \equiv M' & \text{by the definition of } \equiv \text{ with } M \Rrightarrow_* M'',\ M' \Rrightarrow_* M''
\end{array}$$

3.2 Preservation

A fundamental property of a type systems is that evaluation preserves type[13].

We need a number of technical lemmas before we can prove that $\Rrightarrow_*$ preserves types. Proofs are almost always on induction by typing derivations, this allows the context to "grow" with recursive calls while still being well founded by the tree structure of the derivation.

3.2.1 Context Weakening

The following rule is admissible

$$\frac{\Gamma \vdash n : N}{\Gamma, \Gamma' \vdash n : N}$$

by induction on typing derivations

3.2.2 Substitution Preservation

The following rule is admissible[14]

[13] Similar proofs for dependent type systems can be found in Chapter 3 of [Luo94], Section 3.1 of [Miq01](including eta expansion in an implicit system), and in the appendix of [SCA+12]

[14] This lemma is sufficient for our informal account of variable substitution and binding. A fully formal account will be sensitive to the specific binding strategy, and may need to prove this lemma as a corollary from simultaneous substitutions

$$\frac{}{\Diamond \equiv \Diamond}\ \equiv\text{-ctx-empty}$$

$$\frac{\Gamma \equiv \Gamma' \quad M \equiv M'}{\Gamma, x : M \equiv \Gamma', x : M'}\ \equiv\text{-ctx-ext}$$

Figure 6: Contextual Equivalence

$$\frac{\Gamma \vdash n : N \quad \Gamma, x : N, \Gamma' \vdash m : M}{\Gamma, \Gamma' [x := n] \vdash m [x := n] : M [x := n]}$$

by induction on typing derivations

ty-$\star$	$\Gamma, x : N, \Gamma' \vdash \star : \star$	
	$\Gamma, \Gamma' [x := n] \vdash \star : \star$	ty-$\star$
ty-var	$y : M \in \Gamma, x : N, \Gamma'$	
	if $y : M \in \Gamma$, $\quad \Gamma \vdash y : M$	ty-var
	$\Gamma, \Gamma' [x := n] \vdash y : M$	by weakening
	$\Gamma, \Gamma' [x := n] \vdash y [x := n] : M [x := n]$	$x \notin fv(y)$,$x \notin fv(M$
	if $y = x$, $\quad \Gamma \vdash y : N$	
	$\Gamma, \Gamma' [x := n] \vdash y : N$	by weakening
	$N = M$	$y = x$, and context lo
	$\Gamma, \Gamma' [x := n] \vdash y : M$	$x \notin fv(y)$,$x \notin fv(M$
	if $y \in \Gamma'$, $\quad y : M \in \Gamma, x : N, \Gamma'$	
	$y : M [x := n] \in \Gamma, \Gamma' [x := n]$	
	$\Gamma, \Gamma' [x := n] \vdash y : M [x := n]$	ty-var
ty-::	$\Gamma, x : N, \Gamma' \vdash m : M$	
	$\Gamma, \Gamma' [x := n] \vdash m [x := n] : M [x := n]$	by induction
	$\Gamma, \Gamma' [x := n] \vdash m [x := n] :: M [x := n] : M [x := n]$	ty-::
ty-fun-ty	$\Gamma, x : N, \Gamma' \vdash M : \star, \Gamma, x : N, \Gamma', x : M \vdash N : \star$	
	$\Gamma, \Gamma' [x := n] \vdash M [x := n] : \star$	by induction
	$\Gamma, \Gamma' [x := n], y : M [x := n] \vdash N [x := n] : \star$	by induction
	$\Gamma, \Gamma' [x := n] \vdash (y : M [x := n]) \to N [x := n] : \star$	ty-fun-ty
ty-fun	$\Gamma, x : N, \Gamma', f : (x : N) \to M, x : N \vdash m : M$	
	$\Gamma, \Gamma' [x := n], f : (y : N [x := n]) \to M [x := n], y : N [x := n] \vdash m [x := n] : M [x := n]$	by induction
	$\Gamma, \Gamma' [x := n] \vdash \mathsf{fun}\, f\, x \Rightarrow m [x := n] : (x : N [x := n]) \to M [x := n]$	ty-fun
ty-fun-app	$\Gamma, x : N, \Gamma' \vdash m : (x : P) \to M, \Gamma, x : N, \Gamma' \vdash p : P$	
	$\Gamma, \Gamma' [x := n] \vdash p [x := n] : P [x := n]$	by induction
	$\Gamma, \Gamma' [x := n] \vdash m [x := n] : (y : P [x := n]) \to M [x := n]$	by induction
	$\Gamma, \Gamma' [x := n] \vdash m [x := n]\ p [x := n] : M [x := n] [y := p [x := n]]$	ty-fun-app
	$\Gamma, \Gamma' [x := n] \vdash m [x := n]\ p [x := n] : M [y := p, x := n]$	
ty-conv	$\Gamma, x : N, \Gamma' \vdash m : M,\ M \equiv M'$	
	$\Gamma, \Gamma' [x := n] \vdash m [x := n] : M [x := n]$	by induction
	$M \Rrightarrow_* M'',\ M' \Rrightarrow_* M''$	by $\equiv$ -Def
	$M [x := n] \Rrightarrow_* M'' [x := n]$	by $\Rrightarrow_*$ closed under s
	$M' [x := n] \Rrightarrow_* M'' [x := n]$	by $\Rrightarrow_*$ closed under s
	$M [x := n] \equiv M' [x := n]$	$\equiv$ -Def
	$\Gamma, \Gamma' [x := n] \vdash m [x := n] : M' [x := n]$	ty-conv

3.2.3 Context Preservation

When contexts are convertible, typing judgments still hold. We extend the notion of definitional equality to contexts in 6.

the following rule is admissible

$$\frac{\Gamma \vdash n : N \quad \Gamma \equiv \Gamma'}{\Gamma' \vdash n : N}$$

by induction over typing derivations

ty-$\star$	$\Gamma \vdash \star : \star$	
	$\Gamma' \vdash \star : \star$	ty-$\star$
ty-var	$x : M \in \Gamma$	
	$x : M' \in \Gamma',\ M \equiv M'$	by $\Gamma \equiv \Gamma'$
	$\Gamma' \vdash x : M'$	ty-var
	$M' \equiv M$	by symmetry
	$\Gamma' \vdash x : M$	ty-conv
ty-conv	$\Gamma \vdash m : M,\ M \equiv M'$	
	$\Gamma' \vdash m : M$	by induction
	$\Gamma' \vdash m : M'$	ty-conv
ty-::	$\Gamma \vdash m :: M : M$	
	$\Gamma' \vdash m : M$	by induction
	$\Gamma' \vdash m :: M : M$	ty-::
ty-fun-ty	$\Gamma \vdash M : \star, \Gamma, x : M \vdash N : \star$	
	$\Gamma' \vdash M : \star$	by induction
	$\Gamma, x : M \equiv \Gamma', x : M$	$\equiv$-ctx-ext
	$\Gamma', x : M \vdash N : \star$	by induction with $\Gamma, x : M \vdash N : \star$
	$\Gamma' \vdash (x : M) \to N : \star$	ty-fun-ty
ty-fun	$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	
	$\Gamma, f : (x : N) \to M \equiv \Gamma', f : (x : N) \to M$	$\equiv$-ctx-ext
	$\Gamma, f : (x : N) \to M, x : N \equiv \Gamma', f : (x : N) \to M, x : N$	$\equiv$-ctx-ext
	$\Gamma', f : (x : N) \to M, x : N \vdash m : M$	by induction with $\Gamma, f : (x : N) \to M, x : N \vdash m : M$
	$\Gamma' \vdash \mathsf{fun}\, f\, x \Rightarrow m : (x : N) \to M$	ty-fun
ty-fun-app	$\Gamma \vdash m : (x : N) \to M,\ \Gamma \vdash n : N$	
	$\Gamma' \vdash m : (x : N) \to M$	by induction
	$\Gamma' \vdash n : N$	by induction
	$\Gamma' \vdash m\, n : M\, [x \coloneqq n]$	ty-fun-app

3.2.4 Inversion

In the preservation proof we will need to reason backwards about the typing judgments implied by a typing derivation of term syntax. However this induction does not go through directly, and must be weakened up to definitional equality.

Thus we can show this more general rule

$$\frac{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m : P \quad P \equiv (x : N) \to M}{\Gamma, f : (x : N) \to M, x : N \vdash m : M}$$

is admissible. By induction on typing derivations,

ty-fun	$\Gamma, f : (x : N') \to M', x : N' \vdash m : M',\ (x : N') \to M' \equiv (x : N) \to M$	
	$N' \equiv N, \quad M' \equiv M$	by stability of fun-ty
	$\Gamma, f : (x : N') \to M', x : N' \equiv \Gamma, f : (x : N) \to M, x : N$	by reflexivity of $\equiv$, extended with previous equalit
	$\Gamma, f : (x : N) \to M, x : N \vdash m : M'$	by preservation of contexts
	$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	ty-conv
ty-conv	$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m : P',\ P' \equiv P,\ P \equiv (x : N) \to M$	
	$P' \equiv (x : N) \to M$	by transitivity
	$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	by induction
other rules	impossible	the term position has the form $\mathsf{fun}\, f\, x \Rightarrow m$

This allows us to conclude the more strait-forward corollary

$$\frac{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m : (x : N) \to M}{\Gamma, f : (x : N) \to M, x : N \vdash m : M}$$

by noting that $(x : N) \to M \equiv (x : N) \to M$, by reflexivity

3.2.5 $\Rrightarrow$-Preservation

The following rule is admissible

$$\frac{\Gamma \vdash m : M \quad m \Rrightarrow m'}{\Gamma \vdash m' : M}$$

by induction on the typing derivation $\Gamma \vdash m : M$, specializing on $m \Rrightarrow m'$,

ty-$\star$	$\Rrightarrow$-$\star$	$\Gamma \vdash \star : \star$, $\star \Rrightarrow \star$	follows directly
ty-var	$\Rrightarrow$-var	$\Gamma \vdash x : M$, $x \Rrightarrow x$	follows directly
ty-conv		$\Gamma \vdash m : M$, $M \equiv M'$	
	all $\Rrightarrow$	$m \Rrightarrow m'$	
		$\Gamma \vdash m' : M$	by induction
		$\Gamma \vdash m' : M'$	ty-conv
ty-::		$\Gamma \vdash m : M$	
	$\Rrightarrow$-::-red	$m \Rrightarrow m'$	
		$\Gamma \vdash m' : M$	by induction
	$\Rrightarrow$-::	$m \Rrightarrow m'$, $M \Rrightarrow M'$	
		$\Gamma \vdash m' : M$	by induction
		$M \equiv M'$	by promoting $M \Rrightarrow M'$
		$\Gamma \vdash m' : M'$	ty-conv
		$\Gamma \vdash m' :: M' : M'$	ty-::
		$M' \equiv M$	by symmetry
		$\Gamma \vdash m' :: M' : M$	ty-conv
ty-fun-ty		$\Gamma \vdash M : \star$, $\Gamma, x : M \vdash N : \star$	
	$\Rrightarrow$-fun-ty	$N \Rrightarrow N'$, $M \Rrightarrow M'$	
		$\Gamma \vdash M' : \star$	by induction
		$\Gamma, x : M \vdash N' : \star$	by induction
		$M \equiv M'$	by promoting $M \Rrightarrow M'$
		$\Gamma, x : M \equiv \Gamma, x : M'$	by reflexivity of $\equiv$, extended with $M \equiv M$
		$\Gamma, x : M' \vdash N' : \star$	by preservation of contexts
		$\Gamma \vdash (x : M') \to N' : \star$	ty-fun-ty
ty-fun		$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	
	$\Rrightarrow$-fun	$m \Rrightarrow m'$	
		$\Gamma, f : (x : N) \to M, x : N \vdash m' : M$	by induction
		$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m' : (x : N) \to M$	ty-fun
ty-fun-app		$\Gamma \vdash n : N$	
	$\Rrightarrow$-fun-app-red	$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m : (x : N) \to M$, $m \Rrightarrow m'$, $n \Rrightarrow n'$	
		$\mathsf{fun}\, f\, x \Rightarrow m \Rrightarrow \mathsf{fun}\, f\, x \Rightarrow m'$	$\Rrightarrow$-fun
		$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m' : (x : N) \to M$	by induction
		$\Gamma, f : (x : N) \to M, x : N \vdash m'$	by fun-inversion
		$\Gamma \vdash n' : N$	by induction
		$\Gamma \vdash m' \left[f := \mathsf{fun}\, f\, x \Rightarrow m', x := n'\right] : M \left[x := n'\right]$	by typed substitutions (f is not free in M)
		$M \left[x := n'\right] \equiv M \left[x := n\right]$	by substitution by steps, $\equiv$ symmetry
		$\Gamma \vdash m' \left[f := \mathsf{fun}\, f\, x \Rightarrow m', x := n'\right] : M \left[x := n\right]$	ty-conv
	$\Rrightarrow$-fun-app	$\Gamma \vdash m : (x : N) \to M$, $m \Rrightarrow m'$, $n \Rrightarrow n'$	
		$\Gamma \vdash m' : (x : N) \to M$	by induction
		$\Gamma \vdash n' : N$	by induction
		$\Gamma \vdash m'\, n' : M \left[x := n'\right]$	ty-fun-app
		$M \left[x := n'\right] \equiv M \left[x := n\right]$	by substitution by steps, $\equiv$ symmetry
		$\Gamma \vdash m'\, n' : M \left[x := n\right]$	ty-conv

3.3 Progress

The second key lemma is to show preservation, computation is "finished" or that a further step can be taken. For non-dependently typed programming languages, these steps are easy to characterize, but for dependent types there are issues. If we characterize computation with the $\Rrightarrow$ relation, the progress lemma holds in a meaningless way since we can always take a reflexive step. Thus we need a more realistic computation relation, ideally one that is not reflexive, is deterministic and that is a sub relation of $\Rrightarrow_*$. We can choose a call-by-value relation since this meets all the properties required, and is a standard execution strategy, that reflects actual implementations.

values,

$$\begin{array}{lcl} \text{v} & ::= & \star \\ & | & (x : M) \to N \\ & | & \mathsf{fun}\, f\, x \Rightarrow m \end{array}$$

Figure 7: Surface Language Value Syntax

$$\frac{}{(\mathsf{fun}\, f\, x \Rightarrow m)\, v \rightsquigarrow m\, [f := \mathsf{fun}\, f\, x \Rightarrow m, x := v]}$$

$$\frac{m \rightsquigarrow m'}{m\, n \rightsquigarrow m'\, n}$$

$$\frac{n \rightsquigarrow n'}{v\, n \rightsquigarrow v\, n'}$$

$$\frac{m \rightsquigarrow m'}{m :: M \rightsquigarrow m' :: M}$$

$$\frac{}{v :: M \rightsquigarrow v}$$

Values are characterized by the sub-grammar in 7. As usual, functions with any body are values. Additionally the Type universe is a value, and function types [15] are values.

A call-by-value relation is defined in 3.3. The reductions are standard for a call-by-value lambda calculus, except that type annotations are only removed from values.

the following rule is admissible

$$\frac{m \rightsquigarrow m'}{m \Rrightarrow m'}$$

Thus $\rightsquigarrow$ also preserves types.

3.3.1 Canonical forms

We will need a technical lemma that determines the form of a value of function type in an empty context

If $\vdash v\ :\ P$ and $P \equiv (x : N) \to M$ then $v = \mathsf{fun}\, f\, x \Rightarrow m$.

by induction on the typing derivation

ty-fun	$\vdash \mathsf{fun}\, f\, x \Rightarrow m\ :\ (x : N) \to M$	follows immediately
ty-conv	$\vdash v\ :\ P, \vdash v\ :\ P', P \equiv P'$	
	$P' \equiv (x : N) \to M$	by transitivity, symmetry
	$v = \mathsf{fun}\, f\, x \Rightarrow m$	by induction
ty-$\star$	$\vdash \star\ :\ \star, \star \equiv (x : N) \to M$	
	$\star \not\equiv (x : N) \to M$	by the stability of $\equiv$
ty-fun-ty	$\vdash (x : M) \to N\ :\ \star, \star \equiv (x : N) \to M$	
	$\star \not\equiv (x : N) \to M$	by the stability of $\equiv$
other rules	impossible	since they do not type values

as a corollary,

If $\vdash v\ :\ (x : N) \to M$ then $v = \mathsf{fun}\, f\, x \Rightarrow m$.

3.3.2 Progress

Finally we can prove

If $\vdash m\ :\ M$ then m is a value or there exists m' such that $m \rightsquigarrow m'$

As usual this follows form induction on the typing derivation

[15] evaluated to whnf

ty-$\star$	$\vdash \star : \star$		
	$\star$is a value		
ty-var	$\vdash x \,:\, M$		impossible in an empty contex
ty-conv	$\vdash m : M'$, $M' \equiv M$		
	m is a value or there exists m' such that $m \rightsquigarrow m'$		by induction on $\vdash m \,:\, M'$
ty-::	$\vdash m :: M \,:\, M, \vdash m \,:\, M$		
	m is a value or there exists m' such that $m \rightsquigarrow m'$		by induction
	if m is a value,	$m :: M \rightsquigarrow m$	
	if $m \rightsquigarrow m'$,	$m :: M \rightsquigarrow m' :: M$	
ty-fun-ty	$\vdash (x : M) \to N \,:\, \star$		
	$(x : M) \to N$ is a value		
ty-fun	$\vdash \mathsf{fun}\, f\, x \Rightarrow m \,:\, (x : N) \to M$		
	$\mathsf{fun}\, f\, x \Rightarrow m$ is a value		
ty-fun-app	$\vdash m \,:\, (x : N) \to M, \Gamma \vdash n \,:\, N$		
	m is a value or there exists m' such that $m \rightsquigarrow m'$		by induction
	n is a value or there exists n' such that $n \rightsquigarrow n'$		by induction
	if $m \rightsquigarrow m'$,	$m\, n \rightsquigarrow m'\, n$	
	if m is a value, $n \rightsquigarrow n'$,	$m\, n \rightsquigarrow m\, n'$	
	if m is a value, n is a value,	$m = \mathsf{fun}\, f\, x \Rightarrow p$	by canonical forms of functio
		$(\mathsf{fun}\, f\, x \Rightarrow p)\, n \rightsquigarrow p\, [f := \mathsf{fun}\, f\, x \Rightarrow p, x := n]$	

Progress via call-by-value can be seen as a specific sub-strategy of $\Rrightarrow$. An interpreter is always free to take any $\Rrightarrow$, but if it is unclear which $\Rrightarrow$ to take, either it is a value and no further steps are required, or can fall back on $\rightsquigarrow$ until the the outermost constructor has completed.

3.4 Type Soundness

The language has type soundness, well typed terms will never "get stuck" in the surface language. This follows by iterating the progress and preservation lemmas.

3.5 Type checking is impractical

This type system is inherently non-local. No type annotations are ever required to form a derivation. That means it would be up to a type checking algorithm to guess the types of intermediate terms. For instance,

$$\begin{aligned} &\lambda f \Rightarrow \\ &\quad ...f\, 1_c\, true_c \\ &\quad ...f\, 0_c\, 1_c \end{aligned}$$

what should be deduced for the type of f? One possibility is $f : (n : \mathbb{N}) \to n \star (\lambda - \Rightarrow \mathbb{N}_c)\; \mathbb{B}_c \to ....$ But there are infinitely other possibilities. Worse, if there is an error, it may be impossible to localize. To make a practical type checker we need to insist that the user include some type annotations.

4 Bi-directional Surface Language

There are many possible way to localize the type checking process. We could ask that all variable be annotated at binders. This is ideal from a theoretical perspective, since it matches how type contexts are built up.

However note that, our proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ will look like

$\lambda pr : 1_c \doteq_{\mathbb{N}_c} 0_c \Rightarrow pr\; (\lambda n : (C : (\mathbb{N}_c \to \star)) \to C\, 1_c \to C\, 0_c \Rightarrow n \star (\lambda - : \star \Rightarrow Unit_c)\; \perp_c)\; tt_c : \neg 1_c \doteq_{\mathbb{N}_c} 0_c$

Annotating every binding site requires a lot of redundant information. Luckily there's a better way.

4.1 Bidirectional type checking

Bidirectional type checking is a popular form of lightweight type inference, which strikes a good compromise between the required type annotations and the simplicity of the theory, allowing for localized errors [16]. In the usual

[16] [Chr13] is a good tutorial, [DK21] is a survey of the technique

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \overrightarrow{:} M}\ \overrightarrow{ty}\text{-var}$$

$$\frac{}{\Gamma \vdash \star \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-}\star$$

$$\frac{\Gamma \vdash m \overleftarrow{:} M}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-}::$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star}{\Gamma \vdash (x : M) \to N \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-fun-ty}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N}{\Gamma \vdash m\, n \overrightarrow{:} M\,[x := n]}\ \overrightarrow{ty}\text{-fun-app}$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash m \overleftarrow{:} M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \overleftarrow{:} (x : N) \to M}\ \overleftarrow{ty}\text{-}\mathsf{fun}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} M \quad M \equiv M'}{\Gamma \vdash m \overleftarrow{:} M'}\ \overleftarrow{ty}\text{-conv}$$

Figure 8: Surface Language Bidirectional Typing Rules

bidirectional typing schemes, annotations are only required at the top-level, or around a lambda that is directly applied to an argument[17], for example $(\lambda x \Rightarrow x + x)7$ would need to be written $((\lambda x \Rightarrow x + x) :: \mathbb{N} \to \mathbb{N})\, 7$. Since programers rarely write functions that are immediately evaluated, this style of type checking usually only needs top level functions to be annotated[18]In fact, every example in 2 has enough annotations to type check bidirectionally without further information.

This is accomplished by breaking the typing judgments into two mutual judgments:

- **Type Inference** where type information propagates out of a term, $\overrightarrow{:}$ in our notation.
- **Type Checking** judgments where a term is checked against a type, $\overleftarrow{:}$ in our notation.

This allows typing information to flow from the outside in for type checking judgments and inside out for the type inference judgments. When an inference meets a check, a conversion verifies that the types are definitionally equal. This has the advantage of precisely limiting where conversion rule can be used, since conversion checking is usually an efficient part of dependent type checking.

This enforced flow results in a system that localizes type errors. If a type was inferred, it was unique from the term, so it can be used freely. Checking judgments force terms that could have multiple typings in the previous system to have at most one type.

The surface language supports bidirectional type-checking over the pre-syntax with the rules in figure 8. The rules are almost the same as before except that typing direction is now explicit in the judgment.

As mentioned, bidirectional type checking handles higher order inputs very well. For instance, the expression $\vdash (\lambda x \Rightarrow x\, (\lambda y \Rightarrow y)\, 2) \overleftarrow{:} ((\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ checks because $\vdash (\lambda y \Rightarrow y) \overleftarrow{:} (\mathbb{N} \to \mathbb{N})$ and $\vdash 2 \overleftarrow{:} \mathbb{N}$.

Unlike the undirected judgments of the Type Assignment System, the inference rule of the the bidirectional system does not convert, it us unique up to syntax! For example $x : Vec\, 3 \vdash x \overrightarrow{:} Vec\, 3$, but $x : Vec\, 3 \nvdash x \overrightarrow{:} Vec\, (1 + 2)$. This could cause unexpected behavior around function applications if $\Gamma \vdash m \overrightarrow{:} \mathbb{N} \to \mathbb{N}$ then $m\, 7$ will infer, but if the $\to$ from bing in the head position of the type even under another cast, for instance $\Gamma \vdash m \overrightarrow{:} (\mathbb{N} \to \mathbb{N} :: \star)$ then $m\, 7$ will will not infer.

The reverse issue is possible around Check rules against function definitions. For instance $((\lambda x \Rightarrow x) :: \mathbb{N} \to \mathbb{N})$ will infer, but if computation blocks the $\to$ from bing in the head position, inference will be impossible. As in this expression, $((\lambda x \Rightarrow x) :: (\mathbb{N} \to \mathbb{N} :: \star))$ which will not infer.

For these reasons, realistic implementations will often evaluate the types needed for $\overleftarrow{ty} - \mathsf{fun}$, and $\overrightarrow{ty}$-fun-app into weak head normal form [19]. More advanced bidirectional implementations such as Agda

Cite Ulf

even preform unification as part of their bidirectional type checking.

[17] more generally when an elimination reduction is possible

[18] Even in Haskell, with full Hindley-Milner type inference, top level type annotations are encouraged.

[19] as in [Coq96]

More about extending the system so constraint solving can happen under a check judgment

Clearly explain why this is needed for the cast system, annotating every var is combersome, constraint solving is iffy when things may be undecidable

This document opts for the simplest possible presentation, of bidirectional types. There will always be ways to make type inference more powerful, at the cost of complexity.

4.2 The Bidirectional System is Type Sound

It is possible to prove bidirectional type systems are type sound directly[NM05]. But it would be difficult for the system described here since type annotations evaluate away, complicating preservation. Alternatively we can show that a bidirectional typing judgment implies a type assignment system typing judgment.

- if $\Gamma \vdash m \overrightarrow{:} M$ then $\Gamma \vdash m \, : \, M$
- if $\Gamma \vdash m \overleftarrow{:} M$ then $\Gamma \vdash m \, : \, M$

by mutual induction on the bidirectional typing derivations.

Therefore the bidirectional system is also type sound.

4.3 The TAS System is annotatable[20] by the Bidirectional System

Additionally we can show that the bidirectional system does not preclude any computation available in the type assignment system, though annotations may need to be added (or removedFormally

- if $\Gamma \vdash m \, : \, M$ then $\Gamma \vdash m' \overrightarrow{:} M'$, $\Gamma \vdash m' \overleftarrow{:} M'$, $m \equiv m'$ and $M \equiv M'$

by induction on the typing derivation, adding and removing annotations at each step that are convertible with the original m

5 Absent Logical Properties

When type systems are used as logics, it is desirable that

- There exists an empty type that is uninhabited in the empty context, so the system is **logically sound**[21].
- Type checking is decidable.

Neither the TAS system or the Bidirectional systems has these properties[22].

5.1 Logical Unsoundness

The surface language is logically unsound, every type is inhabited.

5.1.1 Every Type is Inhabited (by recursion)

$\mathsf{fun}\, f\, x \Rightarrow f\, x \qquad : \bot_c$

5.1.2 Every Type is Inhabited (by Type-in-type)

It is possible to encode Girard's paradox, producing another source of logical unsoundness. A subtle form of recursive behavior can be built out of Gerard's paradox[Rei89], but this behavior is no worse then the unrestricted recursion already allowed.

[20]also called completeness

[21]also called "consistent"

[22]These properties are usually shown be showing that the computation that generates conversion is normalizing. A proof for a more logical system can be found in Chapter 4[Luo94]

5.1.3 Logical Unsoundness

Operationally, logical unsoundness will be recognized by programmers as non-termination. Non-termination seems not to matter for programming languages in practice. For instance, in ML the type `f : Int−> Int` does not imply the termination of `f 2`. While unproductive non-termination is always a bug, it seems an easy bug to detect and fix when it occurs. In mainstream languages, types help to communicate the intent of termination, even though termination is not guaranteed by the type system. Importantly, no desirable computation is prevented in order to preserve logical soundness. There will never be a way to include all the terminating computations and exclude all the nonterminating computations. A tradeoff must be made, and programmers likely care more about having all possible computations then preventing non-termination. Therefore, logical unsoundness seems suitable for a dependently typed programming language.

While the surface language supports proofs, not every term typed in the surface language is a proof. Terms can still be called proofs as long as the safety of recursion and type-in-type are checked externally. In this sense, the listed example inequalities are proofs, as they make no use of general recursion (so all recursions are well founded) and universes are used in a safe way (universe hierarchies could be assigned). In an advanced implementation, an automated process could supply warnings when constructs are used in potentially unsafe ways. Traditional software testing can be used to discover if there are actual proof bugs. Even though the type system is not logically sound, type checking still eliminates a large class of possible mistakes. While it is possible to make a subtle error, it is easier to make an error in a paper and pencil proofs, or in typeset latex.

Finally by separating non-termination concerns from the core of the theory this architecture is resilient to change. If the termination checker is updated in Coq, there is some chance older proof scripts will no longer type check. With the architecture proposed here, code will always have the same static and dynamic behavior, though some warnings might appear or disappear.

5.2 Type Checking is Undecidable

Given a thunk $f : Unit$ defined in PCF, it can be encoded into the surface system as a thunk $f' : Unit_c$, such that if f reduces to the canonical $Unit$ then $f' \Rrightarrow_* \lambda A.\lambda a.a$

$\vdash \star : f' \star \star$ type-checks by conversion exactly when f halts.

If there is a procedure to decide type checking we can decide exactly when any PCF function halts. Since checking if a PCF function halts is undecidable, type checking here is undecidable.

Again this the root of the problem is the non-termination that results by allowing Turing complete computations, which are apparently necessary for a realistic programing language.

Luckily undecidability of type checking is not as bad as it sounds for several reasons. First, the pathological terms that cause non-terminating conversion are rarely created on purpose. In the bidirectional system, conversion checks will only happen at limited positions, ant it is possible to use a counter to warn or give errors at code positions that do not convert because normalization takes too long. Heuristic methods of conversion checking seem to work well enough in practice even without a counter. It is also possible to embed proofs of conversion directly into the syntax [SCA+12].

Many dependent type systems, such as Agda, Coq, and Lean, aspire to decidable type checking. However these systems allow extremely fast growing functions to be encoded (such as Ackerman's function). A fast growing function can generate a very large index that can be used to check some concrete but unpredictable property, (how many Turing machines whose code is smaller then n halt in n steps?). When this kind of computation is lifted to the type level, type checking is computationally infeasible, to say the least.

Many mainstream programming languages have undecidable type checking. If a language admits a sufficiently powerful macro or preprocessor system that can modify typing, this would make type checking undecidable (this makes the type system of C, C++[23], Scala, and Rust undecidable). Unless type features are considered very carefully, they can often create undecidable type checking (Java generics, C++ templates, Scala implicit parameters[24] and OCaml modules, make type checking undecidable in those languages respectively). Haskell may be the most popular statically typed language with decidable type checking (though GHC compiler flags that make type checking undecidable are popular). Even the Hindley-Milner type checking algorithm that underlies Haskell and ML, has a worst case double exponential complexity, which under normal circumstances would be considered intractable.

In practice these theoretical concerns are irrelevant since programers are not giving the compiler "worst case" code. Even if they did, the worst that can happen is the type checking will time-out in the compilation process. When this happens programmers can fix their code, modify or remove macros, or add typing annotations. Programers

[23]apparently even the grammar of C++ is undecidable
[24]without a maximum search depth

in conventional languages are already entrusted with almost unlimited power to cause harm. Programs regularly delete files, read and modify sensitive information, and send emails (some of these are even possible from within the language's macro systems). Relatively speaking, undecidable type checking is not the biggest concern.

Finally, for the system descried in this thesis, users are expected to use the elaboration procedure defined in the next chapter that will bypass the type checking described here. That elaboration procedures is also undecidable, but only for extremely pathological terms.

6 Related work

6.1 Bad logics, ok programming languages?

Unsound logical systems that work as programming languages go back to at least Church's lambda calculus which was originally intended to be part of a foundation for mathematics[25]. In the 1970s, Martin-Löf proposed a system with type-in-type that was shown logically unsound by Girard (as described in the introduction in [ML72]). In the 1980s, Cardelli explored the domain semantics of a system with general recursive dependent functions and type-in-type, and produced more good examples [Car86].

The first progress and preservation style proof of type soundness for a language with general recursive dependent functions and type-in-type that I am aware of comes form the Trellys Project [SCA+12]. At the time their language had several additional features not included in the surface language. Additionally, the surface language uses a simpler notion of definitional equality resulting in a simpler proof of type soundness. Later work in the Trellys Project[CSW14, Cas14] used modalities to separate terminating and non-terminating fragments of the language, to allow both general recursion and logically sound reasoning. In general, the surface language has been deeply informed by the Trellys project[SCA+12][CSW14, Cas14] [SW15] [Sjö15] and the Zombie language[26] it produced.

6.2 Implementations

Several programming language implementations support this combination of features (though none prove type soundness). Pebble[BL84] was a very early language with dependent types, though conversion did not associate alpha equivalent types [27]. Coquand implemented an early bidirectional algorithm to type-check a language with type-in-type[Coq96]. Cayenne [Aug98] is a Haskell like language that combines dependent types with type-in-type and non-termination. Agda supports general recursion and type-in-type with compiler flags. Idris supports similar "unsafe" features.

6.3 Other Dependent Type Systems

There are many flavors of dependent type systems that are similar in spirt to the language presented here, but maintain logical soundness at the expense of computation.

The Calculus of Constructions (CC, CoC)[CH88] is one of the first minimal dependent type systems. It contains shockingly few rules, but can express a wide variety of constructions via parametric encodings. The systems does not allow type in type, instead type[28] lives in a larger universe $\star : \square$, where $\square$ is not considered a type. Even though the Calculus of Constructions does not allow type-in-type is it is still **impredicative** in the sense that function types can quantify over $\star$ while still being in $\star$. For instance, the polymorphic identity $(X : \star) \to X \to X$ has type $\star$ so the polymorphic identity can be applied to itself. From the perspective of the surface language this impredictivity is modest, but still causes issues in the presence of classical logical assumptionsadapted from examples that were first worked out for the Calculus of Constructions.

Several other systems were developed that directly extended or modified the Calculus of Constructions. The Extended Calculus of Constructions (ECC)[Luo90, Luo94], extends the calculus of constructions with a predicative hierarchy of universes and dependent pair types. The Implicit Calculus of Constructions (ICC)[Miq01, BB08] presents an extrinsic typing system[29], unlike the Type Assignment System presented in this chapter, the Implicit Calculus of Constructions allows implicit qualification over terms in addition to explicit quantification over terms

[25]"There may, indeed, be other applications of the system than its use as a logic."[Church, 1932, p.349, A Set of Postulates for the Foundation of Logic]

[26]https://github.com/sweirich/trellys

[27]according to [Rei89]

[28]called "prop"

[29]Sometimes called "Curry-style", in contrast to intrinsic systems which are sometimes called "Church-style".

(also a hierarchy of universes, and a universe of "sets"). Other extensions to the Calculus of Constructions that are primarily concerned with data (UCC, CIC) will be reviewed in chapter 4.

The lambda cube is a system for relating 8 interesting typed lambda caluli in relation to each other. Presuming terms should always depend on terms, there are 3 additional dimensions of dependency: term depending on types, types dependent on types, and types depending on terms. The simply typed lambda calculus has only term dependency. System F additionally allows Types to depend on types. The Calculus of Constructions has all forms of dependency[30].

Pure Type Systems (PTS)[31] generalize the lambda cube to allow any number of type universe with any forms of dependency. Notably this includes the system with one type universe where type-in-type. Universe hierarchies can also be embedded in a PTS. The system described in this chapter is almost a PTS, except that it contains unrestricted recursion and the method of type annotation is different. All pure type systems such as System F and the Calculus of Constructions have corresponding terms in the the Surface Language, by renaming their type universes into the surface language type universe.

As previously mentioned Martin Lof Type Theory (MTLL) [ML72] is one of the oldest frameworks for dependent type systems. MLTT is designed to be open, so that new constructs can be added with the appropriate introduction, elimination, and typing rules. The base system comes with a predicative hierarchy of universes, and at least dependently typed functions and a propositional equality type. The system has two flavors characterized by it's handling of definitional equality. If types are only identified by convertibility (as every system described so far) it is called Intentional Type Theory (ITT). If the system allows proofs of equality to associate types, it is called Extensional Type Theory. Since MTLL is open ended, the Calculus of Constructions can be added to it as a subsystem[AH04, Hof97a].

References

[AH04] David Aspinall and Martin Hofmann. Dependent types. In *Advanced Topics in Types and Programming Languages*, pages 45–86. MIT Press, 2004.

[Aug98] Lennart Augustsson. Cayenne a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. Association for Computing Machinery.

[BB08] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 365–379, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 1–50, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[Car86] Luca Cardelli. A polymorphic [lambda]-calculus with type: Type. Technical report, DEC System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 1986.

[Cas14] Chris Casinghino. Combining proofs and programs. 2014.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2–3):95–120, February 1988.

[Chr13] David Raymond Christiansen. Bidirectional typing rules: A tutorial. Technical report, 2013.

[Coq96] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.

[30] Recommended reading Chapter 14 [SU06]

[31] previously called "Generalized Type Systems"

[Hof97a] Martin Hofmann. *Extensional constructs in intensional type theory.* Springer Science & Business Media, 1997.

[Hof97b] Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

[JZSW10] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. *SIGPLAN Not.*, 45(1):275–286, January 2010.

[KSW20] Wen Kokke, Jeremy G. Siek, and Philip Wadler. Programming language foundations in agda. *Science of Computer Programming*, 194:102440, 2020.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* 1994.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Miq01] Alexandre Miquel. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 344–359, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[ML72] Per Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.

[NM05] Aleksandar Nanevski and Greg Gregory Morrisett. Dependent type theory of stateful higher-order functions. 2005.

[Pie02] Benjamin C Pierce. *Types and programming languages.* MIT press, 2002.

[Pro13] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[Rei89] Mark B. Reinhold. Typechecking is undecidable when 'type' is a type. Technical report, 1989.

[SCA+12] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *Mathematically Structured Functional Programming*, 76:112–162, 2012.

[Sjö15] Vilhelm Sjöberg. A dependently typed language with nontermination. 2015.

[Smi88] Jan M. Smith. The independence of peano's fourth axiom from martin-lÃ¶f's type theory without universes. *The Journal of Symbolic Logic*, 53(3):840–845, 1988.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* Elsevier, 2006.

[SW15] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 369–382, 2015.

[Tak95] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

7 alt bidirectional formalization

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \overrightarrow{:} M}\ \overrightarrow{ty}\text{-var}$$

$$\frac{}{\Gamma \vdash \star \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-}\star$$

$$\frac{\Gamma \vdash m \overleftarrow{:} M}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-::}$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star}{\Gamma \vdash (x : M) \to N \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-fun-ty}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} C \quad C \Rrightarrow_* (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N}{\Gamma \vdash m\, n \overrightarrow{:} M\, [x := n]}\ \overrightarrow{ty}\text{-fun-app}$$

$$\frac{C \Rrightarrow_* (x : N) \to M \quad \Gamma, f : (x : N) \to M, x : N \vdash m \overleftarrow{:} M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \overleftarrow{:} C}\ \overleftarrow{ty}\text{-}\mathsf{fun}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} M \quad M \equiv M'}{\Gamma \vdash m \overleftarrow{:} M'}\ \overleftarrow{ty}\text{-conv}$$

This is more in the spirit of bidirectional type checking, and closer to the implemented algorithm. If keeping the old style, need to more carefully review the examples since more annotations may be needed. Additionally it makes the symmetry clear against the cast lang of the next section.

However,

- this formalization makes it difficulty to bake in regularity conditions, since the reductions may not preserve bidirectionally. Which cause non-biderecitonal types to be added to the context.
- it seems reasonable to add some restriction to M' in the conversion rule so as not to allow checks at untyped types. though this may subtly interact with other invariants

$$\frac{\Gamma \vdash m \overrightarrow{:} M \quad M \equiv M' \quad \Gamma \vdash M \overleftarrow{:} \star}{\Gamma \vdash m \overleftarrow{:} M'}\ \overleftarrow{ty}\text{-conv}$$

- thought the symmetry with the cast language is more clear, it seems to make proofs more difficult. Though fully proving it might be more enlightening
- This allows a strict super-set of terms from the current formalization, so in principle it is easer to draw conclusions from the existing system

The alt formalization allows the key proofs from this section,

7.1 The bidirectional System is Type Sound

- if $\Gamma \vdash m \overrightarrow{:} M$ then $\Gamma \vdash m : M$
- if $\Gamma \vdash m \overleftarrow{:} M$ then $\Gamma \vdash m : M$

by mutual induction on the bidirectional typing derivations, generating definitional equalities from reduction.

7.2 The Bidirectional System is Conservative

Additionally we can show that the bidirectional system does not preclude any computation available in the type assignment system. Formally

- if $\Gamma \vdash m : M$ then $\Gamma \vdash m' \overleftarrow{:} M$, $m \equiv m'$
- if $\Gamma \vdash m : M$ then $\Gamma \vdash m' \overrightarrow{:} M'$, $m \equiv m'$ and $M \equiv M'$

by induction on the typing derivation, adding and removing annotations at each step that are convertible with the original m

8 alt bidirectional formalization (with some regularity)

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma \vdash m \overleftarrow{:} M}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-::}$$

this variant better motivates the placement of the location positions. however it does not match the undirected type system precisely which may make induction harder, since the undirected system does not have regularity.

This system is still sound, but may not be complete

- $x : 3 \vdash x\ :\ 3$, but not $X : 3 \vdash m \overleftarrow{:} \star$, for any $3 \equiv m$
- $(\vdash \star\ :\ (\lambda - \Rightarrow \star)\,(\star\star)$, since $(\lambda - \Rightarrow \star)\,(\star\star) \equiv \star$ is ok with norm)

it is likely complete over reasonable contexts (though regularity lemmas or assumptions will be needed)

- if $\Gamma \vdash$, $\Gamma \vdash m\ :\ M$ then $\Gamma' \vdash m' \overleftarrow{:} M$, $\Gamma' \overleftarrow{\vdash}$, $m \equiv m'$
- if $\Gamma \vdash$, $\Gamma \vdash m\ :\ M$ then $\Gamma' \vdash m' \overrightarrow{:} M'$, $\Gamma' \overleftarrow{\vdash}$, $m \equiv m'$ and $M \equiv M'$

Alternatively modify the undirected rule to

$$\frac{\Gamma \vdash M : \star \quad \Gamma \vdash m\ :\ M}{\Gamma \vdash m :: M\ :\ M}$$

this will start out unmotivated, but make things cleaner in the long run. regularity assumptions or a system with regularity will be needed.

9 alt bidirectional formalization (with strict regularity)

$$\frac{\Gamma \overleftarrow{ok} \quad x : M \in \Gamma}{\Gamma \vdash x \overrightarrow{:} M}\ \overrightarrow{ty}\text{-var}$$

$$\frac{\Gamma \overleftarrow{ok}}{\Gamma \vdash \star \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-}\star$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma \vdash m \overleftarrow{:} M}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-::}$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star}{\Gamma \vdash (x : M) \to N \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-fun-ty}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} C \quad \Gamma \vdash C \overleftarrow{:} \star \quad C \Rrightarrow (x : N) \to M \quad \Gamma \vdash (x : N) \to M \overleftarrow{:} \star \quad \Gamma \vdash n \overleftarrow{:} N}{\Gamma \vdash m\, n \overrightarrow{:} M\,[x \coloneqq (n :: N)]}\ \overrightarrow{ty}\text{-fun-app}$$

$$\frac{\Gamma \vdash C \overleftarrow{:} \star \quad C \Rrightarrow (x : N) \to M \quad \Gamma \vdash (x : N) \to M \overleftarrow{:} \star \quad \Gamma, f : (x : N) \to M, x : N \vdash m \overleftarrow{:} M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \overleftarrow{:} C}\ \overleftarrow{ty}\text{-}\mathsf{fun}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} M \quad M \equiv M' \quad \Gamma \vdash M' \overleftarrow{:} \star}{\Gamma \vdash m \overleftarrow{:} M'}\ \overleftarrow{ty}\text{-conv}$$

this variant ensures a strict form of regularity, similar to an undirected system that embeds well formed contexts into typing derivations. The advantage here is that a strict form of regularity holds

- $\Gamma \vdash m \overleftarrow{:} M$ then $\Gamma \overleftarrow{\vdash}$ and $\Gamma \vdash M \overleftarrow{:} \star$
- $\Gamma \vdash m \overrightarrow{:} M$ then $\Gamma \overleftarrow{\vdash}$ and $\Gamma \vdash M \overleftarrow{:} \star$

Additionally, well formedness conditions are "baked" into the derivation that makes it available inductively (inductions are clearly well founded).

clearly sound. but completeness may be difficult to show without building a regular version of the type system, additionally it will be difficult to show some WHNF reduction will maintain bidirectionality

- if $\Gamma \vdash$, $\Gamma \vdash m\ :\ M$ then $\Gamma' \vdash m' \overleftarrow{:} M$, $\Gamma' \overleftarrow{\vdash}$, $m \equiv m'$
- if $\Gamma \vdash$, $\Gamma \vdash m\ :\ M$ then $\Gamma' \vdash m' \overrightarrow{:} M'$, $\Gamma' \overleftarrow{\vdash}$, $m \equiv m'$ and $M \equiv M'$

This could be presented in a fancy way that greys out invariants while leaving the algorithmic procedure black

10 TODO

- The proofs may break the flow, review that.
- Should I say more about regularity?
  - tempted to add a summery section with greyed out regularity conditions. Advice expereinced readers to skip ahead. make 2 systems, the irregular system (as written) and the regular system (2 systems will make concluding things from induction easier)
- include references from https://www.lix.polytechnique.fr/~vsiles/papers/PTSATR.pdf
- discuss invariants that don't hold
  - g : (f : nat -> bool) -> (fpr : (x :Nat -> IsEven x -> f x = Bool) -> Bool
  - g f _ = f 2
- in the presence of non terminating proof functions
  - g : (n : Nat) -> (fpn : (x : IsEven n) -> Bool
  - g f _ = f 2
- example of non-terminating functions being equal
- caveat about unsupported features
- go through previous stack overflow questions to remind myself about past confusion.
- make user implementation is smooth around this
- write up style guide
- NuPRl bar types "Constable and Smith had papers at LICS around 1987 or so, leading to both Smith's and Crary's theses on the topic"

Todo list

11 unused

Triangle Property

$\forall m, m'.\, m \Rrightarrow m' \;\to\; m \Rrightarrow max\,(m) \;\wedge\; m' \Rrightarrow max\,(m)$

$$\begin{tikzcd} m \arrow[r, Rrightarrow] \arrow[d, Rrightarrow] & m' \arrow[dl, Rrightarrow] \\ max(m) & \end{tikzcd}$$

$$\forall m, m', n.\ m \Rrightarrow m' \ \wedge\ m \Rrightarrow_* n \ \rightarrow\ \exists n'.\ m' \Rrightarrow_* n' \ \wedge\ n \Rrightarrow n'$$

$$\begin{tikzcd}
& m \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow, "*"] & \\
m' \arrow[dr, Rrightarrow, "*"] & & m \arrow[dl, Rrightarrow] \\
& m' &
\end{tikzcd}$$