Chapter 2 (draft)

October 21, 2021

Part I
Surface language

In an ideal world programmers would write perfect programs whose correctness is fully proven. The **Surface Language** presented in this chapter allows for this ideal, but difficult, workflow. Programmers should "think" in the surface language, and the machinery of later sections should reinforce an understanding of the surface type system, while being transparent to the programmer.

The surface language presented in this chapter is a minimal dependent type system. It will serve both as foundation for further chapters. As much as possible, the syntax use's standard modern notation [1]. The semantics are intended to be as simple as possible and compatible with other well studied intentional dependent type theories [2].

I deviate from a standard dependent type theory to include features to ease programming at the expense of 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 easier when logical soundness has been abandoned. Despite this, type soundness is a achievable and a practical type checking algorithm is given.

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

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, dependent recursive 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. Location data ℓ is marked at every position where a type error might occur.

There is no destination between types and terms in the syntax[3], 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.

Several abbreviations are convenient when dealing with a language like this. These are listed in 2

2 Examples

The surface system is extremely expressive. Several example surface language constructions can be found in 2. I abuse turnstile notion 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 $2_c : \mathbb{N}_c$.

The surface language subsumes or encodes many classic typed lambda calculi. For instance all pure type systems[4] such as System F and the Calculus of Constructions can represent their terms into the Surface Language [5]. Additionally the examples from [Car86] where Cardelli has studied a similar system, can be expressed here.

[1] several alternative syntax exist in the literature. In this document the typed polymorphic identity function be written as $\lambda - \; x \Rightarrow x \; : \; (X : \star) \to X \to X$. In [CH88] it might be written $(\lambda X : \star)\,(\lambda x : X)\,x \; : \; [X : \star]\,[x : X]\,X$, Martin Loff/Hoff (TODO PI notation)

[2] most terms in this chapter could be translated into the calculus of constructions, or other pure type systems, (TODO actually test that these could all be plugged into agda with approrite flags)

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

[4] previously called "generalized type systems"

[5] by renaming their type universes into the Surface type universe

source labels,

ℓ	::=	...	
	|	.	no source label

type contexts,

Γ	::=	$\Diamond \mid \Gamma, x : M$	

expressions,

m, n, M, N	::=	x	variable
	|	$m ::_\ell M$	annotation
	|	$\star$	type universe
	|	$(x : M_\ell) \to N_{\ell'}$	function type
	|	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	|	$m_\ell\, n$	application

Figure 1: Surface Language Syntax

$(x : M) \to N$	written	$M \to N$	when	$x \notin fv\,(N)$
$\mathsf{fun}\, f\, x \Rightarrow m$	written	$\lambda x \Rightarrow m$	when	$f \notin fv\,(m)$
$\ldots\, x \Rightarrow \lambda y \Rightarrow m$	written	$\ldots\, x\, y \Rightarrow m$		
x	written	$-$	when	$x \notin fv\,(m)$ when x binds m
$m ::_\ell M$	written	$m :: M$	when	ℓ is irrelevant
$(x : M_\ell) \to N_{\ell'}$	written	$(x : M) \to N$	when	ℓ, ℓ' are irrelevant
$m_\ell\, n$	written	$m\, n$	when	ℓ is irrelevant

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

Figure 2: Surface Language Abbreviations

	$\vdash \bot_c$	$:= (X : \star) \to X$	$: \star$	Void, "empty" type, lo
	$\vdash Unit_c$	$:= (X : \star) \to X \to X$	$: \star$	Unit, logical true
	$\vdash tt_c$	$:= \lambda - \; x \Rightarrow x$	$: Unit_c$	trivial proposition, po
	$\vdash \mathbb{B}_c$	$:= (X : \star) \to X \to X \to X$		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
$x : \mathbb{B}_c$	$\vdash \neg_c x$	$:= x\, \mathbb{B}_c\, false_c\, true_c$	$: \mathbb{B}_c$	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 numbers
	$\vdash 0_c$	$:= \lambda - \; - z \Rightarrow z$	$: \mathbb{N}_c$	
	$\vdash 1_c$	$:= \lambda - \; s\, z \Rightarrow s\, z$	$: \mathbb{N}_c$	
	$\vdash 2_c$	$:= \lambda - \; s\, z \Rightarrow s\, (s\, z)$	$: \mathbb{N}_c$	
	$\vdash n_c$	$:= \lambda - \; s\, z \Rightarrow s^n\, z$	$: \mathbb{N}_c$	
$x : \mathbb{N}_c, y : \mathbb{N}_c$	$\vdash x +_c y$	$:= \lambda X\, s\, z \Rightarrow x\, X\, s\, (y\, X\, s\, z)$	$: \mathbb{N}_c$	
$X : \star, Y : \star$	$\vdash X \times_c Y$	$:= (Z : \star) \to (X \to Y \to Z) \to Z$	$: \star$	pair, logical and
$X : \star, Y : \star$	$\vdash Either_c\, X\, Y$	$:= (Z : \star) \to (X \to Z) \to (Y \to Z) \to Z$	$: \star$	either, logical or
$X : \star$	$\vdash \neg_c X$	$:= X \to \bot_c$	$: \star$	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 number
$X : \star, Y : X \to \star$	$\vdash \exists_c x : X . Y\, x$	$:= (C : \star) \to ((x : X) \to Y\, x \to C) \to C$	$: \star$	dependent pair, logica
$X : \star, x_1 : X, x_2 : X$	$\vdash x_1 \doteq_X x_2$	$:= (C : (X \to \star)) \to C\, x_1 \to C\, x_2$	$: \star$	Leibniz equality
$X : \star, x : X$	$\vdash refl_{x:X}$	$:= \lambda - \; cx \Rightarrow cx$	$: x \doteq_X x$	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

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[6] are encodable with 2 tags, 0 and successor, where successor also contains the result of the preceding number. So $\mathbb{N}_c$ encodes the possibility of choices, $(A \to A)$ decides how to handle the recursive result of the prior number in the successor case, and the 2nd 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 2 types 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 exactly 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 exactly straight forward. To make the system easier for programmers, data types will be added in chapter 4.

2.2 Predicate encodings

In general we associate the truth value of a proposition with the inhabitation of a type. So, $\bot_c$, 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.

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 would 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"[7]. 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 a term, this is called "Large elimination" can be simulated with type-in-type.

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

For instance, $toLogic$ can convert a $\mathbb{B}_c$ term into it's 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.

$\lambda pr \Rightarrow pr\ (\lambda x \Rightarrow x)\ \bot_c$	:	$\neg_c \star \doteq_\star \bot_c$	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

Note that a proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ is not possible in the Calculus of Constructions[Smi88][8].

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.

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

given the appropriate definitions of $case_c$, $-_c$.

[6] called "church numerals"

[7] 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.

[8] Martin Hofmann excellently motivates the reasoning in [Hof97](incorrect citation) Exercises 2.5, 2.6, 3.7, 3.25, 3.26, 3.43, 3.44

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

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

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

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

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

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

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

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

The rules of the type assignment system are listed in 3.

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. Since the proofs in this section can be done without forcing the context to be well formed, it is tempting to omit them. Here I have compromised by including the extra book keeping in a grey font that will hopefully keep the key ideas in the foreground, while still allowing the most powerful exposition.

language type system the most important property is type soundness, often motivated with the slogan, "well typed programs don't get stuck (in an empty context)"[Mil78][9]. In the setting of TAS systems their is potential for a program to "get stuck" when an argument is given to non-function syntax. 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. The type assignment system is type sound.

Type soundness can be shown with a progress and preservation[10] style proof[WF94]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 avoids a specific class of bad behavior. This type of proof hinges on a suitable definition of the $\equiv$ relation.

The progress/preservation style proof requires $\equiv$ be

- reflexive
- symmetric
- transitive
- closed under (well typed) substitution
- preserves typing
- $\star \not\equiv (x : N) \to M$ does not associate type constructors

it further helps if $\equiv$ is

- closed under normalization

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

$$\frac{\Gamma \vdash m : M \ m \twoheadrightarrow_* \ n \quad \Gamma \vdash m : M \ m' \twoheadrightarrow_* \ n}{\Gamma \vdash m \equiv m' : M} \equiv\text{-Def}$$

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

[10] also called subject reduction, similar proofs in Chapter 3 of [Luo94]

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

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

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

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

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

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

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

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

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

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

Figure 3: Transitive-Reflexive-Closure

- reflexive by definition
- symmetric automatically
- transitive if $\Rrightarrow_*$ is confluent
- closed under substitution if $\Rrightarrow_*$ is closed under substitution
- preserves types, if $\Rrightarrow_*$ preserves types
- $\star \not\equiv (x : N) \to M$ does not associate type constructors since $(x : N) \to M \not\Rrightarrow_* \star$
- closed under normalization automatically

Parallel reductions are defined to make those properties easier to prove.

While this is a simple definition of equality and reduction, others choices are possible, for instance it is possible to extend the relation with contextual information, type information, or even explicit proofs of equality as in ETT. It is also common to assume the properties of $\equiv$ hold without proof.

3.1 Equality

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

The following rule is admissible,

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

by induction on the syntax of m

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

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

is admissible.

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

The following rule is admissible for every substitution σ

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

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, $\sigma\left(x\right) \Rrightarrow \tau\left(x\right)$, written $\sigma \Rrightarrow \tau$

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

by induction on the $\Rrightarrow$ relation.

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

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

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

is admissible.

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

Confluent[11]

By defining normalization with parallel reductions we can show confluence using the methods shown in [Tak95][12]. First define a function max that takes the maximum possible parallel step, such that if $m \Rrightarrow m'$ then $m' \Rrightarrow max\left(m\right)$ and $m \Rrightarrow max\left(m\right)$, referred to as the triangle property.

$$\begin{array}{lcll}
max(\quad (\mathsf{fun}\, f\, x \Rightarrow m)_\ell\ n &) = & max\left(m\right)\left[f := \mathsf{fun}\, f\, x \Rightarrow max\left(m\right), x := max\left(n\right)\right] & \text{otherwise} \\
max(\quad x &) = & x & \\
max(\quad m ::_\ell M &) = & max\left(m\right) & \\
max(\quad \star &) = & \star & \\
max(\quad (x : M_\ell) \to N_{\ell'} &) = & (x : max\left(M\right)_\ell) \to max\left(N\right)_{\ell'} & \\
max(\quad \mathsf{fun}\, f\, x \Rightarrow m &) = & \mathsf{fun}\, f\, x \Rightarrow max\left(m\right) & \\
max(\quad m_\ell\, n &) = & max\left(m\right)_\ell\ max\left(n\right) &
\end{array}$$

$m \Rrightarrow max\left(m\right)$
by induction on the cases of max.
if $m \Rrightarrow m'$ then $m' \Rrightarrow max\left(m\right)$
by induction on the derivation $m \Rrightarrow m'$, with the only interesting cases are where a reduction is not taken

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

it follows that
if $m \Rrightarrow m'$, $m \Rrightarrow m''$, implies $m' \Rrightarrow max\left(m\right)$,$m'' \Rrightarrow max\left(m\right)$, referred to as the diamond property
since the function max will pick a unique term.
the diamond property implies the confluence of $\Rrightarrow_*$
by repeated application of the diamond property
it follows that $\equiv$ is transitive

3.2 Context Lemmas

3.2.1 Typed Substitution

For any $\Gamma \vdash x : N$, the following rule is admissible

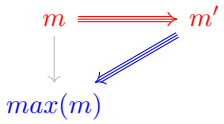
$$\frac{\Gamma, x : N, \Gamma' \vdash m : M}{\Gamma, \Gamma'\left[x := n\right] \vdash m\left[x := n\right] : M\left[x := n\right]}$$

by induction on typing derivations

[11] also "Church-Rosser"
[12] also well presented in [KSW20]

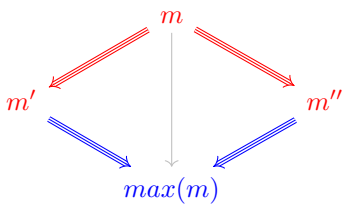
Triangle Property

$\forall m, m'.\ m \Rrightarrow m' \ \rightarrow\ m' \Rrightarrow max\,(m)$



Diamond Property

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



Confluence

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

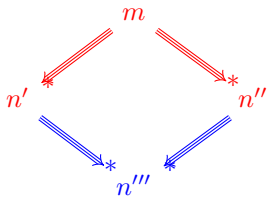


Figure 4: Rewriting Diagrams

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

ty-$\star$		
	$\Gamma' \vdash \star : \star$	ty-$\star$
ty-var	$y : M \in \Gamma$	
	$\Gamma \vdash y : M$	ty-var
	$\Gamma, \Gamma' \vdash y : M$	by weakening [1]
	$y = x$	
	$\Gamma \vdash y : N$	by assumption
	$\Gamma, \Gamma' \vdash y : N$	by weakening [1]
	$N = M$	$y = x$, and con
	$\Gamma, \Gamma' \vdash y : M$	
	$y \in \Gamma'$	
	$y : M \in \Gamma, x : N, \Gamma'$	
	$y : M\,[x \coloneqq n] \in \Gamma, \Gamma'\,[x \coloneqq n]$	
	$\Gamma \vdash y : M\,[x \coloneqq n]$	ty-var
ty-::	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n] : M\,[x \coloneqq n]$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n] :: M\,[x \coloneqq n] \;:\; M\,[x \coloneqq n]$	ty-::
ty-fun-ty	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash M\,[x \coloneqq n] \;:\; \star$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n], y : M\,[x \coloneqq n] \vdash N\,[x \coloneqq n] \;:\; \star$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash (y : M\,[x \coloneqq n]) \to N\,[x \coloneqq n] \;:\; \star$	ty-fun-ty
ty-fun	$\Gamma, \Gamma'\,[x \coloneqq n], f : (y : N\,[x \coloneqq n]) \to M\,[x \coloneqq n], y : N\,[x \coloneqq n] \vdash m\,[x \coloneqq n] \;:\; M\,[x \coloneqq n]$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash \mathsf{fun}\, f\, x \Rightarrow m\,[x \coloneqq n] \;:\; (x : N\,[x \coloneqq n]) \to M\,[x \coloneqq n]$	ty-fun
ty-fun-app	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash p\,[x \coloneqq n] \;:\; P\,[x \coloneqq n]$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n] \;:\; (y : P\,[x \coloneqq n]) \to M\,[x \coloneqq n]$	by induction
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n]\; p\,[x \coloneqq n] \;:\; M\,[x \coloneqq n]\,[y \coloneqq p\,[x \coloneqq n]]$	ty-fun-app
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n]\; p\,[x \coloneqq n] \;:\; M\,[y \coloneqq p, x \coloneqq n]$	
ty-conv		
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n] : M\,[x \coloneqq n]$	by induction
	$\Gamma, x : N, \Gamma' \vdash M \equiv M' : \star$	by assumption
	$M \Rrightarrow_* M''$, $M' \Rrightarrow_* M''$, ...	by $\equiv$-Def
	$M\,[x \coloneqq n] \Rrightarrow_* M''\,[x \coloneqq n]$	by $\Rrightarrow_*$ closed u
	$M'\,[x \coloneqq n] \Rrightarrow_* M''\,[x \coloneqq n]$	by $\Rrightarrow_*$ closed u
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash M\,[x \coloneqq n] \equiv M\,[x \coloneqq n]' : \star$	$\equiv$-Def
	$\Gamma, \Gamma'\,[x \coloneqq n] \vdash m\,[x \coloneqq n] : M'\,[x \coloneqq n]$	ty-conv

3.2.2 Substitution by steps

The following rule is admissible

$$\frac{n \Rrightarrow_* n' \quad \Gamma \vdash m\,[x \coloneqq n] : M}{\Gamma \vdash m\,[x \coloneqq n] \equiv m\,[x \coloneqq n'] : M}$$

since

$n \Rrightarrow_* n'$	by assumption
$m \Rrightarrow_* m$	by refl
$m\,[x \coloneqq n] \Rrightarrow_* m\,[x \coloneqq n']$	by sub by steps[17]
$m\,[x \coloneqq n'] \Rrightarrow_* m\,[x \coloneqq n']$	by refl
$\Gamma \vdash m\,[x \coloneqq n] \equiv m\,[x \coloneqq n'] : M$	$\equiv$-Def

3.2.3 Context Equivalence

3.2.4 Context Preservation

the following rule is admissible

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

by induction over typing derivations, by mutual induction with

[13] unproven
[14] unproven
[15] reflexive corollary
[16] reflexive corollary
[17] unproven

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

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

Figure 5: Contextual Equivalence

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

ty-$\star$	$\Gamma' \vdash$	by $\Gamma \equiv \Gamma'$ [18]
	$\Gamma' \vdash \star : \star$	ty-$\star$
ty-var	$\Gamma' \vdash$	by $\Gamma \equiv \Gamma'$ [19]
	$x : M \in \Gamma$	by assumption
	$x : M' \in \Gamma'$, $\Gamma \vdash M \equiv M' : \star$	by $\Gamma \equiv \Gamma'$
	$\Gamma' \vdash x : M'$	ty-var
	$\Gamma \vdash M' \equiv M : \star$	by symmetry
	$\Gamma' \vdash x : M$	ty-conv
ty-conv	$\Gamma \vdash m : M$	by assumption
	$\Gamma' \vdash m : M$	by induction
	$\Gamma \vdash M \equiv M' : \star$	by assumption
	$\Gamma' \vdash M \equiv M' : \star$	by mutual induction
	$\Gamma' \vdash m : M'$	ty-conv
ty-::	$\Gamma' \vdash m : M$	by induction
	$\Gamma \vdash m :: M : M$	ty-::
ty-fun-ty	$\Gamma' \vdash M : \star$	by induction
	$\Gamma, x : M \equiv \Gamma', x : M$	...
	$\Gamma', x : M \vdash N : \star$	by induction...
	$\Gamma' \vdash (x : M) \to N : \star$	ty-fun-ty
ty-fun	$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	by assumption
	$\Gamma \vdash (x : N) \to M : \star$	by regularity
	$\Gamma' \vdash (x : N) \to M : \star$	by induction
	$\Gamma \equiv \Gamma'$	by assumption
	$\Gamma, f : (x : N) \to M \equiv \Gamma', f : (x : N) \to M$	by context extention
	$\Gamma, f : (x : N) \to M \vdash N : \star$	by regularity
	$\Gamma', f : (x : N) \to M \vdash N : \star$	by induction
	$\Gamma, f : (x : N) \to M, x : N \equiv \Gamma', f : (x : N) \to M, x : N$	by context extension
	$\Gamma', f : (x : N) \to M, x : N \vdash m : M$	by induction
	$\Gamma' \vdash \mathsf{fun}\, f\, x \Rightarrow m : (x : N) \to M$	ty-fun
ty-fun-app	$\Gamma' \vdash m : (x : N) \to M$	by induction
	$\Gamma' \vdash n : N$	by induction
	$\Gamma' \vdash m\, n : M\,[x \coloneqq n]$	ty-fun-app
$\equiv$ -Def	$\Gamma \vdash n : N$	by assumption
	$\Gamma' \vdash n : N$	by mutual induction
	$\Gamma \vdash n' : N$	by assumption
	$\Gamma' \vdash n' : N$	by mutual induction
	$\Gamma' \vdash n \equiv n' : N$	$\equiv$ -Def

3.3 Stability(which section?)

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

by induction on $\Rrightarrow_*$

[18] TODO
[19] TODO

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

Therefore the following rule is admissible

$$\frac{\Gamma \vdash (x : N) \to M \equiv (x : N') \to M' : \star}{\Gamma \vdash N \equiv N' : \star \quad \Gamma, x : N' \vdash M \equiv M' : \star}$$

$\Gamma \vdash (x : N) \to M : \star$, $\Gamma \vdash (x : N') \to M' : \star$, $(x : N) \to M \Rightarrow_* P$, $(x : N') \to M' \Rightarrow_* P$	by expending the definition of $\equiv$
$P = (x : N'') \to M''$, $N \Rightarrow_* N''$, $M \Rightarrow_* M''$, $N' \Rightarrow_* N''$, $M' \Rightarrow_* M''$	by the lemma above
$\Gamma \vdash N : \star$, $\Gamma, x : N \vdash M : \star$	by fun-ty inversion [20]
$\Gamma \vdash N' : \star$, $\Gamma, x : N' \vdash M' : \star$	by fun-ty inversion [21]
$\Gamma \vdash N \equiv N' : \star$	by the definition of $\equiv$
$\Gamma, x : N \equiv \Gamma, x : N'$	by context refl, extended
$\Gamma, x : N' \vdash M' : \star$	by the preservation of ctx.
$\Gamma, x : N' \vdash M \equiv M' : \star$	by the definition of $\equiv$

3.4 Inversions(which section?)

TODO explanation

we can show this more general rule

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

is admissible. By induction,

ty-fun	$\Gamma, f : (x : N') \to M', x : N' \vdash m \,:\, M'$	by assumption
	$\Gamma \vdash (x : N') \to M' \equiv (x : N) \to M : \star$	by assumption
	$\Gamma \vdash N' \equiv N : \star$, $\Gamma, x : N \vdash M' \equiv M : \star$	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 equalities
	$\Gamma, f : (x : N) \to M, x : N \vdash m \,:\, M'$	by preservation of contexts
	$\Gamma, f : (x : N) \to M, x : N \vdash m : M$	ty-conv
ty-conv	$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \,:\, P'$	by assumption
	$\Gamma \vdash P \equiv P' : \star$	by assumption
	$\Gamma \vdash P' \equiv (x : N) \to M : \star$	by assumption[22]
	$\Gamma \vdash P \equiv (x : N) \to M : \star$	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 rule

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

by noting that

$\Gamma \vdash (x : N) \to M \equiv (x : N) \to M : \star$, by reflexivity

3.5 Preservation

3.5.1 $\Rightarrow$-Preservation

The following rule is admissible

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

can now be proven by induction on the typing derivation $\Gamma \vdash m : M$, specializing on $m \Rightarrow m'$,

[20] fwd

[21] fwd

[22] clean up some of these assumptions?

values,

$$
\begin{array}{lcl}
\mathrm{v} & ::= & \star \\
& | & (x : M_\ell) \to N_{\ell'} \\
& | & \mathsf{fun}\, f\, x \Rightarrow m
\end{array}
$$

Figure 6: Surface Language Value Syntax

ty-$\star$	$\Rightarrow$-$\star$	$\Gamma \vdash \star : \star$	follows directly
ty-var	$\Rightarrow$-var	$\Gamma \vdash x : M$	follows directly
ty-conv	all $\Rightarrow$	$m \Rightarrow m'$	
		$\Gamma \vdash m' : M$	by induction
		$\Gamma \vdash M \equiv M' : \star$	by assumption
		$\Gamma \vdash m' : M'$	ty-conv
ty-::	$\Rightarrow$-::-red	$m \Rightarrow m'$	
		$\Gamma \vdash m' : M$	by induction
	$\Rightarrow$-::	$m \Rightarrow m'$, $M \Rightarrow M'$,	
		$\Gamma \vdash m' : M$	by induction
		$\Gamma \vdash m' :: M' : M'$	ty-::
		$\Gamma \vdash M : \star$	by regularity [23]
		$\Gamma \vdash M' : \star$	by induction[24]
		$\Gamma \vdash M' \equiv M : \star$	by promoting $M \Rightarrow M'$, symmetry
		$\Gamma \vdash m' :: M' : M$	ty-conv
ty-fun-ty	$\Rightarrow$-fun-ty	$N \Rightarrow N'$, $M \Rightarrow M'$	
		$\Gamma \vdash M' : \star$	by induction
		$\Gamma, x : M \vdash N' : \star$	by induction
		$\Gamma \vdash M : \star$	by assumption
		$\Gamma \vdash M \equiv M' : \star$	by promoting $M \Rightarrow M'$
		$\Gamma, x : M \equiv \Gamma, x : M'$	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	$\Rightarrow$-fun	$m \Rightarrow m'$	
		$\Gamma, f : (x : N) \to M, x : N \vdash m' : M$	by induction
		$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m' : (x : N) \to M$	ty-fun
ty-fun-app	$\Rightarrow$-fun-app-red	$m \Rightarrow m'$, $n \Rightarrow n'$	
		$\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m' : (x : N) \to M$	by induction
		$\Gamma, f : (x : N) \to M, x : N \vdash m'$	by fun-inversion
		$\Gamma \vdash n' : N$	by induction
		$\Gamma \vdash m' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M [x \coloneqq n']$	by substitutions (f is not free in M)
		$\Gamma \vdash n : N$	by assumption
		$\Gamma \vdash M [x \coloneqq n] \equiv M [x \coloneqq n'] : \star$	by substitution by steps
		$\Gamma \vdash M [x \coloneqq n'] \equiv M [x \coloneqq n] : \star$	by $\equiv$ symmetry
		$\Gamma \vdash m' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M [x \coloneqq n]$	ty-conv
	$\Rightarrow$-fun-app	$m \Rightarrow m'$, $n \Rightarrow n'$	
		$\Gamma \vdash m' : (x : N) \to M$	by induction
		$\Gamma \vdash n' : N$	by induction
		$\Gamma \vdash m'\, n' : M [x \coloneqq n']$	ty-fun-app
		$\Gamma \vdash n : N$	by assumption
		$\Gamma \vdash M [x \coloneqq n] \equiv M [x \coloneqq n'] : \star$	by substitution by steps
		$\Gamma \vdash M [x \coloneqq n'] \equiv M [x \coloneqq n] : \star$	by $\equiv$ symmetry
		$\Gamma \vdash m' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M [x \coloneqq n]$	ty-conv

3.6 Progress

The 2nd key lemma in a progress and preservation proof is to show that, computation is "complete" or that a further computation be taken. For non dependently typed programming languages, these steps are easy to characterize, but for dependent types there are issues. If we characterize the parstep relation as the computation used in progress, the lemma holds in a meaningless way since we can always take a reflexive step. Thus we need a non-reflexive stepping relation, ideally one that is deterministic and that is a sub relation of $\Rightarrow_*$. We can choose Call-by-value relation since this meets all the properties required, and is a standard execution strategy. A deterministic reflexive evaluation strategy exists.

[23] not proven

[24] this use of induction is well founded, TODO

$$\frac{}{(\mathsf{fun}\, f\, x \Rightarrow m)_\ell\, v \rightsquigarrow m\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m, x \coloneqq v]}$$

$$\frac{m \rightsquigarrow m'}{m_\ell\, n \rightsquigarrow m'_\ell\, n}$$

$$\frac{n \rightsquigarrow n'}{v_\ell\, n \rightsquigarrow v_\ell\, n'}$$

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

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

Values are characterized by the sub-grammar in 6. As usual, functions with any body are values. Additionally the Type universe is a value, and function types [25] are values.

A call-by-values relation is defined in 3.6. The reductions are standard for a call-by-value lambda calculus, except that type annotations are removed from values.

the following rules are admissible

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

Thus it is is preservation preserving.

3.7 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.

3.8 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

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

Annotating every binding site requires a lot of redundant annotations. Luckily there's a better way, Bi-directional type checking.

4.1 Bi-directional type checking

is a popular form of lightweight type inference, and strikes a good compromise between the required type annotations and the simplicity of the theory, allowing for localized errors [26]. In the usual bidirectional bidirectional typing scheme annotations are only required at the top-level, or around a lambda that is applied to an argument[27]. Since programers rarely write functions that are immediately evaluated, this style of type checking usually only needs top level functions to be annotated[28]. In fact,

[25] evaluated to whnf

[26] [Chr13] is a good tutorial, [DK21] is a survey

[27] more generally when an elimination reduction is possible

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

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

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

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

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

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

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

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

Figure 7: Surface Language Bidirectional Typing Rules

every example in 2 has enough annotations to type-check bidirectionally as written.

This is accomplished by breaking typing judgments into two forms:

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

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

This enforced flow results in a system that localizes type information. If a type was inferred it was unique, so it can be used freely. Checking judgments localize their their check, even if the underlying term could support multiple types.

The surface language supports bidirectional type-checking over the pre-syntax with the rules in figure 7. These rules are the same as before but the information flow of typing is now explicit in the judgment.

4.2 The bidirectional system is type sound

It is possible to prove bidirectional systems are type sound directly[NM05], but would seem to require constraining the syntax in unrealistic ways[29]. Alternatively we can show by mutual induction 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.

4.3 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 $\Gamma \vdash m' \overrightarrow{:} M$, $\Gamma \vdash m' \overleftarrow{:} M$, and $m \equiv m'$

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

5 Absent Logical Properties

When type systems are considered as a logic, it is desirable that

- There exists an empty type that is uninhabited in the empty context, it is **logically sound**[30].

[29] by dividing the syntax explicitly into introduction and elimination forms

[30] also called "consistent"

- The conversion system normalizes.
- Type checking is decidable.

The surface system has non of these properties[31].

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.

I am unaware of anyone accidentally encountering a falsehood from type-in-type.

5.1.3 logical unsoundness

While the surface language supports proofs, not every term typed in the surface language is a proof.

Logical soundness seems not to matter in programming practice. For instance, in ML the type $\texttt{f} : \texttt{Int} -> \texttt{Int}$ 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. Therefore, logical unsoundness seems suitable for a dependently typed programming language.

The most popular .... Forcing logical soundness incurs a real cost...

Terms can still be called proofs as long as the safety of recursion and type-in-type are monitored externally. In this sense the inequalities listed are proofs, they make no use of general recursion and match implementations from CIC such that universe hierarchies could be assigned. External means can still verify the desired properties, an automated process could supply warnings when unsafe constructs are used, traditional software testing can be used to discover proof bugs. Even though the system is not logically sound, neither are informal paper and pencil proofs.

This architecture is resilient to change. Where a change in the termination checking code of Coq might cause a proof script to no longer run, here the code will always behave in the same way.

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.

Undecidability of type checking is not as bad as it sounds for several reasons. First, the pathological terms that cause normalization are rarely created on purpose. In the bidirectional system, conversion checks will only happen at limited positions, it is possible to use a counter to warn or give errors at code that does not convert because normalization takes too long. Heuristic methods of normalization seem to work well enough in practice even without a counter. It is also possible to embed proofs of conversion directly into the syntax as in[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). This large function can check some concrete but difficult property, (how many Turing machines less than n halt in n steps?). When this kind of computation is lifted to the type level, type checking is computationally infeasible.

Many mainstream type systems have undecidable. If a language admits a Turing complete macro or preprocessor system that can modify typing, this would make type checking undecidable (this would technically make the type system of C, C++, Scala, and Rust undecidable). Unless type features are considered very carefully, they can often create undecidable type checking (Java generics and OCaml modules, make type checking undecidable). Haskell may be the most popular language with with a static type system that allows decidable type checking (it is often more convenient to use GHC compiler flags that make type checking undecidable). Even the Hindley-Milner type checking algorithm that underlies Haskell and ML, has a double exponential complexity.

[31] These properties are usually shown be showing that the computation that generates conversion is normalizing. Some proofs can be found in Chapter 4[Luo90, Luo94]

In practice these theoretical concerns are not born out since the worst that can happen is the type checking can time-out. When this happens programmers can fix their macros, or add typing annotations. Programers in conventional languages are already entrusted with almost unlimited power to to cause harm, programs regularly delete files, read and modify sensitive information, and send emails. Relatively speaking, undecidable type checking is not a 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 are acceptable programming languages go back to at least Church's lambda calculus which was originally intended to be part of a foundation for mathematics. 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[Car86].

The first direct proof of type soundness for a language with general recursive dependent functions, Type-in-Type, and dependent data that I am aware of came form the Trellys Project [SCA+12]. At the time their language had several additional features not included in my surface language. Additionally, my surface language uses a simpler notion of equality and dependent data resulting in an arguably 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 base language has been deeply informed by the Trellys project[SCA+12][CSW14, Cas14] [SW15] [Sjö15] and the Zombie language[32] 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 [33]. Coquand implemented an early bidirectional algorithm to type-check a language with Type-in-Type[Coq96]. Cayenne [Aug98] is a Haskell like language that combined dependent types with Type-in-Type, data 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][34] 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 in modern presentations $\star : \Box$ where $\Box$ 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 int $\star$. 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 assumptions. Many of the examples form this chapter are adapted from examples that appeared along with the Calculus of Constructions.

Several other Systems were developed that directly extended the or modify 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.

CIC, ICC

Coq and Lean trace their core theory back to the Calculus of Constructions.

Frameworks, MLTT (ETT) (ITT) (interpreting CC ML that book chapter), Pure type systems

References

[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.

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

[33] according to [Rei89]

[34] TODO modern treatment in Lectures on the Curry howard isomorphism

[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.

[Hof97] Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

[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.

[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.

[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.

[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 TODO

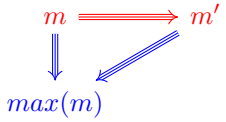
- remove location date from this section entirely!
- fully esperate ctx well formedness judgments
- include references from https://www.lix.polytechnique.fr/~vsiles/papers/PTSATR.pdf
- discuss
  - g : (f : nat -> bool) -> (fpr : (x :Nat -> IsEven x -> f x = Bool) -> Bool
  - g f _ = f 2

- 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
- what is the deal with bidirecitonal type checking?!?!
- caveat about unsupported features
- go through previous stack overflow questions to remindmyself about past confusion.
- make usre implementation is smooth around this
- talk about non-terminaiton
- write up style guide

8 unused

Triangle Property

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



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