Chapter 2 (draft)

October 1, 2021

Part I
Surface language

In an ideal world programmers would write perfect code whose correctness is perfectly proven. The **Surface Language** models this ideal, but difficult, system. 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 Formal Surface Language

The syntax is in figure 1. There is no syntactic destination between types and terms, as is common in full-spectrum systems. However, I will follow the convention that capital letters are used in positions that are intended as types, and lowercase letters are used when the expression may be a term. Location data ℓ is marked at every position where a type error might occur.

2 Examples

The surface system is extremely expressive.

[1]several alternative syntax exist in the literature, (TODO weird french bracket notation) , Martin hoffmen (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)

source labels,			
ℓ			
variable contexts,			
Γ	$::=$	$\Diamond \mid \Gamma, x : M$	
expressions,			
m, n, M, N	$::=$	x	variable
	$\mid$	$m ::_{\ell} M$	annotation
	$\mid$	$\star$	type universe
	$\mid$	$(x : M_{\ell}) \to N_{\ell'}$	function type
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	$\mid$	$m_{\ell}\, n$	application
values,			
v	$::=$	$x \mid \star$	
	$\mid$	$(x : M_{\ell}) \to N_{\ell'}$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	

Figure 1: Surface Language Pre-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	$f \notin fv\,(m)$
$\lambda x \Rightarrow \lambda y \Rightarrow m$	written	$\lambda x\, y \Rightarrow m$		
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

	$\vdash$	$\bot_c$	$:=$	$(x : \star) \to x$	$:$	$\star$	Void, empty type, logical False
	$\vdash$	$Unit_c$	$:=$	$(A : \star) \to A \to A$	$:$	$\star$	Unit, logical True
	$\vdash$	tt_c	$:=$	$\lambda -\ a \Rightarrow a$	$:$	$Unit_c$	trivial proposition, polymorphic identity
	$\vdash$	$\mathbb{B}_c$	$:=$	$(A : \star) \to A \to A \to A$	$:$	$\star$	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
$x : \mathbb{B}_c, y : \mathbb{B}_c$	$\vdash$	$x \,\&_c\, y$	$:=$	$\lambda A \Rightarrow x\, A\ (y\, A)\ (false_c\, A)$	$:$	$\mathbb{B}_c$	boolean and
$x : \mathbb{B}_c, y : \mathbb{B}_c$	$\vdash$	$x \,\&_c\, y$	$:=$	$\lambda A\, then\, else \Rightarrow x\, \mathbb{B}_c\, y\, else$			BAD boolean and
	$\vdash$	$\mathbb{N}_c$	$:=$	$(A : \star) \to (A \to A) \to A \to A$	$:$	$\star$	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	$:=$	$\lambda -\ s\, z \Rightarrow s^n\, z$	$:$	$\mathbb{N}_c$	
$x : \mathbb{N}_c, y : \mathbb{N}_c$	$\vdash$	$x +_c y$	$:=$	$\lambda A\, s\, z \Rightarrow x\, A\, s\ (y\, A\, s\, z)$	$:$	$\mathbb{N}_c$	
$X : \star$	$\vdash$	$\neg_c X$	$:=$	$x \to \bot_c$	$:$	$\star$	logical negation
$X : \star, x_1 : X, x_2 : X$	$\vdash$	$x_1 \doteq_X x_2$	$:=$	$(C : (X \to \star)) \to C\, x_1 \to C\, x_2$	$:$	$\star$	Leibniz equality
$X : \star, x : X$	$\vdash$	$refl_{x:X}$	$:=$	$\lambda -\ cx \Rightarrow cx$	$:$	$x \doteq_X x$	

2.1 Church encodings

Data types are expressible using Church encodings, (in the style of System F). Church encodings embed the elimination principle of data into continuations. So for instance Booleans data is eliminated against true and false, 2 tags with no additional data. This is more recognizable as an if-then-else construct. So $\mathbb{B}_c$ encodes the possibility of choices, $true_c$ picks the first branch, and $false_c$ picks the false branch.

Natural numbers are encodable against 2 tags, 0 and s, where s also contains one smaller number. So $\mathbb{N}_c$ encodes the possibility of choices, $(A \to A)$ decides how to recursively handle the prior number in the s case, and the 2nd argument specifies how to handle the false branch. This can be viewed as a simple looping construct with temporary storage.

2.2 Predicate encodings

With dependent types, logical predicates can be encoded (in the style of Calculus of Constructions).

2.2.1 Leibniz equality

2.3 Large Eliminations

"Large eliminations" can be simulated with type-in-type.

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

Note that such functions are not possible in the Calculus of Constructions, and is used to motivate the extension to the Calculus of Inductive 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
$\lambda pr \Rightarrow pr\ (\lambda b \Rightarrow b\ \star\ Unit_c\ \bot_c)\ tt_c$	$:$	$\neg true_c \doteq_{\mathbb{B}_c} false_c$	
$\lambda pr \Rightarrow pr\ (\lambda n \Rightarrow n\ \star\ (\lambda - \Rightarrow Unit_c)\ \bot_c)\ tt_c$	$:$	$\neg 1_c \doteq_{\mathbb{N}_c} 0_c$	

Note that a proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ is not possible in the Calculus of Constructions[Smi88][3].

[3] Martin Hofmann gives a more semantic proof in ... and excellently motivates the reasoning in [Hof97](incorrect citation) Exercises 2.5, 2.6, 3.7, 3.25, 3.26, 3.43, 3.44

2.4 Recursion

2.4.1 $(x : \mathbb{N}_c) \to 0_c +_c x =_{\mathbb{N}_c} x +_c 0_c$ (by recursion)

$\mathsf{fun}\, f\, x \Rightarrow x\ (0_c +_c x =_{\mathbb{N}_c} x +_c 0_c)\ f\ (refl_{0_c:\mathbb{N}_c}) \qquad : (x : \mathbb{N}_c) \to 0_c +_c x =_{\mathbb{N}_c} x +_c 0_c$

TODO: check and discuss, structural recursion

2.5 logical unsoundness and Nontermination

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

2.5.1 Every type is inhabited (by recursion)

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

2.5.2 Every type is inhabited (by Type-in-type)

It is possible to encode Gerard'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.

2.5.3 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 `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. By the haling problem there will never be a way to include all the terminating computations and exclude all the nonterminating computations. 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.

2.6 Further examples

There are more examples in [Car86] where Cardelli has studied a similar system. All Pure Type Systems[4] can translate into the Surface Language by accumulating their type universes into the surface type universe.

3 Surface Language Type Assignment System

The type assignment system is type sound, "well typed programs don't get stuck". This can be shown with a progress and preservation style proof, with 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

[4]previously called "Generalized type systems"

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

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

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

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

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

$$\frac{\Gamma \vdash m \,:\, M \quad \Gamma \vdash M \equiv M' : \star}{\Gamma \vdash m \,:\, M'} conv$$

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

$$\frac{}{x \Rrightarrow x}$$

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

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

$$\frac{}{\star \Rrightarrow \star}$$

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

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{(\mathsf{fun}\, f\, x \Rightarrow m)_\ell\, n \Rrightarrow m'\,[f := \mathsf{fun}\, f\, x \Rightarrow m', x := n']}$$

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

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

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

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

- reflexive if $\Rrightarrow_*$ is reflexive
- 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 ITT. It is also common in the literature to assume the properties of $\equiv$ hold without proof.

3.1 Equality

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

The following rule is admissible,

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

by induction
it follows that,

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

is admissible. And follows that

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

is admissible.

3.1.2 $\Rrightarrow$, $\Rrightarrow_*$, $\equiv$ is closed under substitutions

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

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

by induction

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

is admissible. And follows that

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

is admissible.

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

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

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

$m \Rrightarrow\ max\,(m)$
by TODO
$m \Rrightarrow\ m'$, $m' \Rrightarrow\ max\,(m)$
by TODO
it follows that
$m \Rrightarrow\ m'$, $m \Rrightarrow\ m''$, implies $m' \Rrightarrow\ max\,(m)$,$m'' \Rrightarrow\ max\,(m)$, referred to as the diamond property, $\Rrightarrow$ is confluent.
it follows that
$\Rrightarrow_*$ is confluent
by TODO
it follows that
$\equiv$ is transitive

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

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

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

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

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

3.2 Context

3.3 Preservation

3.4 Progress

the following rules are admissible

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

Thus it is is preservation preserving and we can use the

3.5 Type Soundness

The language has type soundness, well typed terms will never "get stuck" in the surface language.

3.6 Type checking is undecidable

Given a thunk $f : Unit$ defined in pcf, it can be encoded into the surface system as a thunk $f' : Unit$, 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

3.7 typing is non-local

TODO refer to the specific judgment

$\lambda x.x$ has many types

TODO and therefore untenable

4 Bi-directional Surface Language

4.1 Annotate all the vars

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 this is good since it will be easy to put variables on context.

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

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

This strategy requires a lot of redundant annotations. Luckily there's a better way, Bi-directional type checking.

4.2 Bi-directional

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 ([DK21] is a good survey). This style of type checking usually only needs top level functions to be annotated[5]. In fact, every example in this chapter has enough annotations to type-check under a bidirectional system.

The surface language supports bidirectional type-checking over the pre-syntax with the rules in figure 2. This is accomplished by breaking typing judgments into 2 forms:

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

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

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

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

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

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

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

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

Figure 2: Surface Language Bidirectional Typing Rules

- 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.

Inferences can be turned into checked judgments with an explicit equality check. This precisely limits the use of the problematic conv rule.

4.3 If it types in the bidirectional system then it types in the TAS system

...

4.4 If it types in the TAS system annotations can be added such that an equivalent term types in the bidirectional system

...

4.5 Type-checking in the Bi-directional system is still undecidable

Type checking remains undecidable because of general recursion and type-in-type. However, since the user is not expected to type-check their program directly this should not cause any issues in practice. Even decidable type-checking in dependent type theory is computationally intractable.

4.6 Bi-directional errors are local

...

4.7 Still undecidable

Unfortunately, the system is logically unsound (every type is trivially inhabited with recursion), since our language attempts to be more oriented to programs than proofs. We expect this is acceptable.

5 Implementation

Implemented in Haskell. We have mechanized the type soundness of the type assignment system (without location data) in Coq.

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 a logical 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[KSEI+12][SCA+12][CSW14, Cas14] [SW15] [Sjö15] and the Zombie language[6] it produced.

Several implementations support this combination of features without proofs of type soundness. Coquand presented an early bidirectional algorithm to type-check a similar language [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.

A similar "partial correctness" criterion for dependent languages with non-termination run with Call-by-Value is presented in [JZSW10].

6.2 relation to other formal systems

6.3 relation to other implementations

7 TODO

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?!?!

contact guy about presentations.

caveat about unsupported features

go through prevous stack overflow questions to remindmyself about past confusion.

make usre implementation is smooth around this

talk about non terminaiton

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.

[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.

[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.

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

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

[KSEI$^+$12] Garrin Kimmell, Aaron Stump, Harley D Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 15–26, 2012.

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

[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.