Subtyping with Power Types*

David Aspinall
LFCS, University of Edinburgh, U.K.
http://www.dcs.ed.ac.uk/home/da

May 2000

Abstract

This paper introduces a typed λ-calculus called λ_{Power}, a predicative reformulation of part of Cardelli's *power type* system. Power types integrate subtyping into the typing judgement, allowing bounded abstraction and bounded quantification over both types and terms. This gives a powerful and concise system of dependent types, but leads to difficulty in the meta-theory and semantics which has impeded the application of power types so far. Basic properties of λ_{Power} are proved here, and it is given a model definition using a form of applicative structures. A particular novelty is the auxiliary system for *rough typing*, which assigns simple types to terms in λ_{Power}. These "rough" types are used to prove strong normalization of the calculus and to structure models, allowing a novel form of containment semantics without a universal domain.

1 Introducing Power Types

Power types were introduced in a seminal paper by Cardelli [9]. The notion is that $\mathsf{Power}(A)$ is a type "whose elements are all of the subtypes of the type A,"

$$\frac{A\ \mathsf{type}}{\mathsf{Power}(A)\ \mathsf{type}}$$

In place of a separate definition of subtyping, a relation between types is induced by inhabitation of power types, so

$$A \leq B \quad =_{\text{def}} \quad A : \mathsf{Power}(B)$$

The rules for power types are chosen to make this definition sensible. Cardelli called the three basic rules power introduction, elimination and subtyping:

$$\frac{A\ \mathsf{type}}{A : \mathsf{Power}(A)} \qquad \frac{M : A \qquad A : \mathsf{Power}(B)}{M : B} \qquad \frac{A : \mathsf{Power}(B)}{\mathsf{Power}(A) : \mathsf{Power}(\mathsf{Power}(B))}$$

*WARNING! Draft 1.30. This version is presently incomplete, and work is in progress on an improved revised version. Full proofs are included here as a matter of record, they will not part be of a submitted version. Please send me any questions or comments.

The first rule makes the induced subtyping relation reflexive. The second rule is the characteristic subtyping rule of *subsumption*, which adds subtype polymorphism to the system. The third rule expresses monotonicity of the *Power* operator, and together with the second rule, it makes the induced subtyping relation transitive. Other rules capture the subtyping behaviour of type constructors.

Cardelli meant his type system to be used for programming languages with object-oriented features. Power types can encode *bounded* type abstraction and quantification used in OOP with the usual λ-abstraction and dependent function space,

$$\begin{aligned} \Lambda\alpha \leq A.M &\quad =_{\text{def}} \quad \lambda\alpha{:}\mathsf{Power}(A).M \\ \forall\alpha \leq A.B &\quad =_{\text{def}} \quad \Pi\alpha{:}\mathsf{Power}(A).B \end{aligned}$$

This is a simplification, since there is no need to add new constructs.

Bounded type abstraction and quantification appear in early attempts to capture the polymorphism of functions (or methods) which operate on subclasses of a particular class, where a class is modelled by a type and the class hierarchy is modelled by the subtype relation. Whereas the function $\Lambda\alpha{:}\mathsf{Type}.M$ may be applied to any type argument A, the function $\Lambda\alpha \leq B.M$ can only be applied to a type A which is a subtype of B. For further explanation, see [21]. (The work described here grew from a slightly different application: in ASL+ [2], subtyping models specification refinement, and $\Lambda X \leq SP.M$ is a parameterised specification which can be applied to any refinement of SP.)

As well as power types, Cardelli's 1988 system has impredicative polymorphism via the $\mathsf{Type} : \mathsf{Type}$ axiom; it also has dependent sum and product types, recursive types, variant types, record types and abstract types. The system is so powerful, in fact, that when viewed as a logic it is inconsistent — every type is inhabited in the empty context — and type-checking is undecidable. This was no surprise; Cardelli intended his system as a showcase combining most of the typing ideas that he and others had studied in the preceding decade. After 1988, various fragments of the full system were studied, notably in the type system of Quest [11, 8]. Instead of power types, Quest used power *kinds* to capture subtyping, where $\mathsf{Power}(A)$ is the kind of all subtypes of A, and does not enjoy the status of a type itself. One reason for this restriction was semantic: it was difficult to incorporate a type of all subtypes into the models of polymorphism being studied for Quest at the time.

As far as I know, power types have not been studied extensively since Cardelli's work; this is perhaps the first in-depth study. First I define a calculus called λ_{Power} (Section 3). It is almost a fragment of Cardelli's system, except for a richer power introduction rule and an equality judgement. Then I give some brief examples (Section 2), before considering the meta-theory (Section 4) and a semantics (Sections 6 and 7). The semantics and some of the meta-theory are based on *rough typing*, a way of assigning "rough" non-dependent types to λ_{Power} terms (Section 5). Finally, Section 9 summarizes.

2 Examples in λ_{Power}

As a calculus of functions, λ_{Power} is no more expressive than the simply-typed λ-calculus.[1] In contrast with Cardelli's system, it is predicative: we cannot write a function which

[1] If M is typable in λ_{Power}, then the type-erasure of M can be assigned a simple type, treating Π and *Power* as families of constants. This is demonstrated indirectly by the "rough" typing rules in Section 5.

operates on any type, so there is no System F style universal polymorphism. All type operators are parameterised on subtypes of a given type.

Despite this, λ_{Power} can express complex typings, because of the powerful combination of dependent types and arbitrarily nested power types. In particular, the calculus goes beyond λP, the type system of the Edinburgh Logical Framework, and beyond $\lambda P_{\leq}$, the extension of λP with subtyping introduced by [3]. I describe a few examples next. A brief programming-based example is given in Section 2.1; an example of using power types for higher-order subtyping in in Section 2.2, and an example of using λ_{Power} as a logical framework in Section 2.3. The use of λ_{Power} for ASL+ is described elsewhere [2].

2.1 A simple programming example

Here's a quick programming-style example in λ_{Power} to demonstrate the use of dependent types with subtypes.

Suppose *int* is an atomic type and let Γ_{PERM} be the context:

$$\begin{aligned} \mathit{nat} &: \mathit{Power}(\mathit{int}), \\ \mathit{Upto} &: \mathit{nat} \to \mathit{Power}(\mathit{nat}), \\ \mathit{Perm} &: \Pi n{:}\mathit{nat}.\ \mathit{Power}((\mathit{Upto}\ n) \to (\mathit{Upto}\ n)) \\ \mathit{Invperm} &: \Pi n{:}\mathit{nat}.\ (\mathit{Perm}\ n) \to (\mathit{Perm}\ n) \end{aligned}$$

Imagine that $\mathit{Upto}\ n$ stands for the set $\{\, m \in \mathit{nat} \mid m \leq n \,\}$, and $\mathit{Perm}\ n$ is the set of permutations of $\{\, 1, \ldots, n \,\}$, which is a subset of the set of functions from $\mathit{Upto}\ n$ to $\mathit{Upto}\ n$. The function $\mathit{Invperm}\ n\ p$ yields the inverse of the permutation p on such a set. Here is a function to apply the inverse of a permutation of $\{\, 1, \ldots, n \,\}$ to a number in that range:

$$\mathit{ApplyPerm} =_{\text{def}} \lambda n{:}\mathit{nat}.\, \lambda p{:}\mathit{Perm}\ n.\, \lambda m{:}\mathit{Upto}\ n.\ \mathit{Invperm}\ n\ p\ m$$

Using subsumption for $\mathit{Invperm}\ n\ p$, we can get the expected typing:

$$\Gamma_{\text{PERM}} \rhd \mathit{ApplyPerm} : \Pi n{:}\mathit{nat}.\, (\mathit{Perm}\ n) \to (\mathit{Upto}\ n) \to (\mathit{Upto}\ n).$$

which reveals that $\mathit{ApplyPerm}\ n\ f$ is in fact a function from $\mathit{Upto}\ n$ to $\mathit{Upto}\ n$.

2.2 Subtyping type operators and families

Systems of *higher-order subtyping* extend subtyping to type-constructors. The prototypical one is $F^{\omega}_{\leq}$ [10], in which one can declare a type variable ranging over type operators,

$$F \quad \leq \quad (\lambda\beta \leq \mathit{nat}.\, \mathit{List}(\beta \times \beta)).$$

A system with dependent types instead of polymorphism is $\lambda P_{\leq}$ [3], in which one can declare a variable ranging over type families,

$$G \quad \leq \quad (\lambda x{:}\mathit{nat}.\, \mathit{Vec}_{\mathit{nat}}(5 * x)).$$

In the first case, F ranges over constructors that map a subtype β of *nat* to a subtype of $\mathit{List}(\beta \times \beta)$; in the second case G ranges over constructors that map an element x of *nat* to a subtype of the type of vectors of numbers with $5 * x$ elements.

Both systems have a pointwise rule for subtyping operators and a corresponding application rule:

$$\frac{\Gamma,\ \alpha : K \ \triangleright\ A \leq B}{\Gamma \ \triangleright\ \lambda\alpha{:}K.\, A \leq \lambda\alpha{:}K.\, B} \qquad (\text{SUB-}\lambda)$$

$$\frac{\Gamma \ \triangleright\ H \leq J \qquad \Gamma \ \triangleright\ J\, C : K}{\Gamma \ \triangleright\ H\, C \leq J\, C} \qquad (\text{SUB-APP})$$

The second premise of (SUB-APP) ensures that the application $J\,C$ is well-typed; this implies that $H\,C$ is also well-typed. Here's an example using (SUB-APP):

$$\frac{\begin{array}{lcl} G\,n & \leq & (\lambda x{:}\mathsf{nat}.\, \mathsf{Vec}_{\mathsf{nat}}(5 * x))\, n \\ (\lambda x{:}\mathsf{nat}.\, \mathsf{Vec}_{\mathsf{nat}}(5 * x))\, n & \leq & \mathsf{Vec}_{\mathsf{nat}}(5 * n) \end{array}}{G\,n \quad \leq \quad \mathsf{Vec}_{\mathsf{nat}}(5 * n)}$$

(where $n : \mathsf{nat}$ in the context). This is derived using conversion and transitivity.

In a semantics where $[\![A \leq B]\!]$ is interpreted as a relation between $[\![A]\!]$ and $[\![B]\!]$ (for example, as containment of PERs), to interpret the rule (SUB-λ) we must lift the subtype relation to type functions in a pointwise fashion. This is the basis of the interpretation suggested in [7] sometimes called the *HOPER model*.

In λ_{Power}, there is no rule directly corresponding to (SUB-λ). Indeed it is impossible to prove anything with the form $\Gamma \ \triangleright\ \lambda\alpha{:}K.\, A : \mathsf{Power}(C)$. The rules above are hard to interpret semantically, because the interpretation of $\lambda\alpha{:}K.\, A : \mathsf{Power}(C)$ must be considered pointwise rather than as a subset inclusion, so the meaning of *Power* would depend on its context in a term. The subtyping relation here is really a family of many relations.

Perhaps surprisingly, power types can express similar typings without the pointwise rules. Suppose that F is a subtype of a type-constructor H with domain K; this is like asking F to be an element of $\Pi\alpha{:}K.\ \mathsf{Power}(H\,\alpha)$, since each application $F\,M$ must be a subtype of $H\,M$. This "η-like" expansion for Π-types works uniformly and we can declare: Higher-order bounded abstraction and quantification, and also bounded dependent type functions, can all be reduced to abstraction and quantification over Π-types with power types in their codomain. Instead of the declarations above, in λ_{Power} we can write:

$$\begin{array}{lcl} F & : & \Pi\beta{:}\mathsf{Power}(\mathsf{nat}).\ \mathsf{Power}(\mathsf{List}(\beta \times \beta)) \\ G & : & \Pi x{:}\mathsf{nat}.\ \mathsf{Power}(\mathsf{Vec}_{\mathsf{nat}}\,(5 * x)) \end{array}$$

To derive $G\,n \leq \mathsf{Vec}_{\mathsf{nat}}(5 * n)$ we need only one use of ordinary application:

$$\frac{G : \Pi x{:}\mathsf{nat}.\ \mathsf{Power}(\mathsf{Vec}_{\mathsf{nat}}\,(5 * x)) \qquad n : \mathsf{nat}}{G\,n : \mathsf{Power}(\mathsf{Vec}_{\mathsf{nat}}\,(5 * n))}$$

Substitution in the application rule for dependent products takes place of conversion and transitivity needed before, so derivations in λ_{Power} can be more direct.[2] I believe that this is an original observation about power types, which provides some justification for using power types[3]

[2] But *practical* effects on type-checking algorithms have not been investigated yet.

[3] Luca Cardelli told me that the use for Π-types with *Power* in the codomain, was unclear. Higher-order subtyping is a direct application of them. Since the work described here was undertaken, Crary has also used this idea in his power kind system [14].

2.3 λ_{Power} as a logical framework

λ_{Power} is related to λP, which underlies the Edinburgh LF [22]. One idea behind the LF project was that, given a framework for defining logics, it should be possible to develop *generic* computer-assisted proof tools which work for any logic encoded in the framework. But to use such proof tools in practice, carefully-manufactured forms of abbreviation are vital, to manage the burden of formality.

Adding subtyping to LF introduces two useful forms of abbreviation:

- shorter and better representations of object logic syntax, and
- proof reuse: one proof term can prove several similar propositions.

The desire for subtyping in LF was first mentioned by [29], to improve the representation for Hoare Logic. Later, [32] showed examples of proof-reuse through a restricted form of subtyping called *refinement types*. More recently, [3] proved that the direct addition of subtyping to λP has a decidable type-checking problem. To achieve the benefits of subtyping in other type theories for proof assistants, [28] advocates the addition of subtyping to a typed variant of Martin-Löf's logical framework. Several other studies of subtyping in dependent type theories are currently underway.

Here is a sketch of how λ_{Power} can be used as a logical framework with rich forms of subtyping. It's quite easy to see that λ_{Power} can be used in the same way as λP. Let υ be an atomic type. Then declare a universe of types by writing

$$U \;=_{\text{def}}\; \mathit{Power}(\upsilon).$$

We can use U in place of Type in LF, to declare the term formers and judgements of a logic. If $\Gamma \rhd A : U$ and $\Gamma, x : A \rhd B : U$, then we do *not* have $\Gamma \rhd \Pi x{:}A.\, B : U$, but rather $\Gamma \rhd \Pi x{:}A.\, B : \mathit{Power}(\Pi x{:}A.\, \upsilon)$. Since λP lacks quantification or abstraction over types, this difference has little effect, and we can translate any λP judgement into one which holds in λ_{Power}.[4]

With power types we can declare one syntactic category to be a subtype of another, or one judgement to be a subtype of another, so that every proof of the first judgement is also a proof of the second. This is also possible in the proposals studied in [32, 3], but λ_{Power} goes beyond both these systems by allowing refinements of the universe U itself.

Gardner proposed doing this [16] to help adequacy proofs. She defined a framework ELF+ which distinguishes between terms that represent: object-level syntax, proof terms, and other terms. This partitioning allows more precise statements of adequacy theorems, which state that the terms and judgements of the logic are in correspondence with their representations in the framework. I don't know whether this advantage can be inherited in λ_{Power}; one would need to study properties of canonical typings of terms, or a translation semantics [6]. Nevertheless, it is instructive to look at the "emulation" of ELF+ inside λ_{Power}, revisiting some of Gardner's examples to see where the addition of power types is useful. To emulate ELF+ in λ_{Power}, declare three subtypes:

$$\begin{aligned} \mathtt{Sort} &: \mathit{Power}(U) \\ \mathtt{Judge} &: \mathit{Power}(U) \\ \mathtt{Type} &: \mathit{Power}(U) \end{aligned}$$

[4] Perhaps, moreover, λ_{Power} is conservative over λP under this translation. I haven't checked this.

(In fact, we could equally well begin with three atomic types). Types inhabiting $\mathtt{Sort}$ will be the syntactic categories of the object logic, whilst types inhabiting $\mathtt{Judge}$ will be judgements. (As usual, the representation of the object logic is made by encoding its term constructors and rules as dependently typed constants and types, respectively.) The universe $\mathtt{Type}$ houses "extra constants" which are artifacts of the encoding with no correspondence in the object logic, and Π-abstractions of sorts and judgements.

An encoding where power types are useful is higher-order logic (HOL).

Higher-order logic. The LF representation of higher-order logic (HOL) is given in [4]. Simple types τ of the form ι, o, and $\tau \Rightarrow \tau$ are encoded in an LF type $\mathtt{dom} : \mathtt{Type}$, with

$$\begin{aligned} \mathtt{dom} &: \mathtt{Type} \\ \mathtt{i} &: \mathtt{dom} \\ \mathtt{o} &: \mathtt{dom} \\ \Rightarrow &: \mathtt{dom} \to \mathtt{dom} \to \mathtt{dom} \\ \mathtt{obj} &: \mathtt{dom} \to \mathtt{Type} \end{aligned}$$

HOL terms with domain τ are represented as elements of $\mathtt{obj}(\tau)$. ELF+ improves this, showing $\mathtt{dom}$ and $\mathtt{obj}$ to be artifacts of the encoding, inhabiting $\mathtt{Type}$, and giving $\mathtt{obj}$ the type

$$\mathtt{obj} : \mathtt{dom} \to \mathtt{Sort}$$

showing that elements of $\mathtt{obj}(\tau)$ correspond to object logic syntax.

But in both LF and ELF+, the proliferation of $\mathtt{obj}$ quickly pollutes large terms. In λ_{Power}, we can remove it altogether and declare

$$\mathtt{dom} : \mathit{Power}(\mathtt{Sort})$$

The mapping $\mathtt{obj}$ is now implicit; the representation of the logic becomes more concise, yet no less accurate. For example, the application term former becomes:

$$\mathtt{app} : \Pi s, t{:}\mathtt{dom}.\ (s \Rightarrow t) \to s \to t$$

instead of

$$\mathtt{app} : \Pi s, t{:}\mathtt{dom}.\ \mathtt{obj}(s \Rightarrow t) \to \mathtt{obj}(s) \to \mathtt{obj}(t).$$

Although simple, it is important to emphasise that this example goes beyond many other proposals [32, 3], because $\mathtt{dom}$ is declared as a sub-*kind* of the universe $\mathtt{Sort}$. Power types apply uniformly; other systems would have to be extended with sub-kinding to cope with this example. For more elaborate examples, other systems even need special extensions for higher-order sub-kinding.

Modal logic S4. The representation of Hilbert-style S4 in LF, due to [4], uses an auxiliary judgement Φ *valid* to define the intended judgement Φ *true*. The auxiliary judgement is used because the rule for necessity introducing $\Box$ should only apply to true judgements under no assumptions; this would not follow were the rule encoded in the usual way for a single judgement Φ *true*.

In ELF+, the example proceeds by declaring:

$$\begin{aligned} \mathtt{o} &: \mathtt{Sort} \\ \supset &: \mathtt{o} \to \mathtt{o} \to \mathtt{o} \\ \Box &: \mathtt{o} \to \mathtt{o} \\ \mathtt{true} &: \mathtt{o} \to \mathtt{Judge} \\ \mathtt{valid} &: \mathtt{o} \to \mathtt{Type} \\ \mathtt{C} &: \Pi\Phi{:}\mathtt{o}.\, \mathtt{valid}(\Phi) \to \mathtt{true}(\Phi) \end{aligned}$$

The ELF+ typing of `valid` indicates that an expression $\mathtt{valid}(\Phi)$ is an artifact of the encoding since it inhabits `Type` rather than `Judge`. The pseudo-rule `C` expresses the relationship between the auxiliary judgement and `true`.

In λ_{Power}, we can instead declare:

$$\mathtt{valid} : \Pi\Phi{:}\mathtt{o}.\, \mathit{Power}(\mathtt{true}(\Phi))$$

Now the use of the rule `C` becomes implicit, and no longer pollutes the encoded proof terms. Moreover, one proof term can prove several related judgements involving both `valid` and `true`, but I shall not give examples here. (A general explanation is given by the logical understanding of subtyping as intuitionistic implication, studied by [26].) A similar example by [32] shows that the cut-free proofs can be considered as a subtype of all natural deduction proofs for first-order logic.

The S4 example is also possible in $\lambda P_{\leq}$. Like the HOL example, with subtyping the "artifact" universe `Type` is no longer needed. However, more complicated examples might still need the extra universe `Type` to hold auxiliary judgements used to prove side-conditions of rules in the object logic, for example.

3 The System λ_{Power}

First comes the context-free abstract syntax. Let $\boldsymbol{V}$ be a fixed countable infinite set of variables and $\mathcal{K}$ be a set of atomic type constants. The set $\boldsymbol{T}_{\mathcal{K}}$ of *pre-terms* is given by:

$$\boldsymbol{T} \;::=\; \mathcal{K} \;\mid\; \boldsymbol{V} \;\mid\; \lambda\boldsymbol{V}{:}\boldsymbol{T}.\,\boldsymbol{T} \;\mid\; \boldsymbol{T}\,\boldsymbol{T} \;\mid\; \Pi\boldsymbol{V}{:}\boldsymbol{T}.\,\boldsymbol{T} \;\mid\; \mathit{Power}(\boldsymbol{T})$$

(writing $\boldsymbol{T}$ as short for $\boldsymbol{T}_{\mathcal{K}}$). shall use these metavariable conventions:

- variables: $x, y, \ldots \in \boldsymbol{V}$
- atomic types: $\kappa, \ldots \in \mathcal{K}$
- pre-terms: $A, B, C, D, E, F, \ldots, M, N, P, \ldots \in \boldsymbol{T}$

I use usual conventions for writing pre-terms in the abstract syntax: applications associate to the left; the scope of bound variables extends as far right as possible; parentheses are used for grouping. The notions of closed term, free variables of a term $FV(M)$, substitution $M[N/x]$, and β, η reduction and conversion are all defined as usual. Terms which are α-equivalent are considered as syntactically identical; the symbol $\equiv$ is reserved for syntactic

identity. The dependent product $\Pi x{:}A.\,B$ degenerates to the ordinary function space $A \to B$ when x does not appear free in B. An abbreviation is sometimes used for repeated abstraction or quantification over the same domain; for example, $\Pi x, y{:}A.\,B$ stands for the term $\Pi x{:}A.\,\Pi y{:}A.\,B$.

A *pre-context* is a sequence of variable declarations $x_1 : A_1, x_2 : A_2 \ldots$ where no variable is declared more than once. The empty pre-context is sometimes written $\langle\rangle$, otherwise it is invisible. I use Γ and variants to range over pre-contexts.

Not all pre-terms make sense. The well-formed pre-terms consist of *terms* and *types*, defined in Definition 3.1 below. These are not disjoint; types are also terms of the calculus. I often use letters near the beginning of the alphabet to range over pre-terms which turn out to be types.

3.1 Presentation of λ_{Power}

Terms and types are defined via three judgement forms:

$\rhd \Gamma$	Γ is a well-formed context
$\Gamma \rhd M : A$	In context Γ, M has type A
$\Gamma \rhd M = N : A$	In context Γ, M and N are equal at type A

These judgements are defined simultaneously by the rules shown in Figures 1 and 2. As usual, I shall write $\Gamma \rhd M : A$ to mean "the judgement $\Gamma \rhd M : A$ is derivable," *etc.* The system λ_{Power} is close to a predicative fragment of Cardeilli's original system [9]; the difference is that we use an equality judgement in the presentation, and the more powerful (REFL).

Here is a brief outline of the rules.

Context formation (Figure 2.)
These rules are standard. The rule (EMPTY) says that the empty context is valid and (EXTEND) adds declarations to contexts. The judgement $\Gamma \rhd A : \mathsf{Power}(B)$ serves to say that A is a well-formed type, as well as asserting that A is a subtype of B. This is a general pattern.

Typing rules (Figure 1.)
The rules (VAR), (λ), (APP), (SUB) and (CONV) are standard. The rule (CONV) is also standard, and ensures that equal types have the same elements. The rule (ATOMIC) introduces atomic types; each atomic type is a subtype of itself, so is self-evidently well-formed.

The rule-scheme (REFL) is novel, it expands to this:

$$\frac{\Gamma \rhd M : \Pi x_1{:}A_1.\ \ldots \Pi x_n{:}A_n.\ \mathsf{Power}(B)}{\Gamma \rhd M : \Pi x_1{:}A_1.\ \ldots \Pi x_n{:}A_n.\ \mathsf{Power}(M\,x_1 \cdots x_n)}$$

Reflexivity of subtyping for types is the case that $n = 0$. For each $n > 0$, the rule (REFL) asserts reflexivity of subtyping for n-ary type-valued functions[5] (the example in Section 2.2 motivates this).

The rule (Π) generalises the usual contravariant subtyping rule for function spaces to dependent products. The last premise is a well-formedness check that the term B can be typed under the assumption $x : A$.

[5] A technical note: (REFL) adds a case of η-subject reduction to the sytem; if $y : \Pi x{:}A.\ \mathsf{Power}(B)$ then with (λ) we get $\lambda x{:}A.\ y\,x : \Pi x{:}A.\ \mathsf{Power}(y\,x)$, but we need (REFL) to get $y : \Pi x{:}A.\ \mathsf{Power}(y\,x)$.

$$\frac{\rhd\ \Gamma}{\Gamma\ \rhd\ \kappa : \mathsf{Power}(\kappa)} \quad (\textsc{atomic})$$

$$\frac{\rhd\ \Gamma \qquad x \in Dom(\Gamma)}{\Gamma\ \rhd\ x : \Gamma(x)} \quad (\textsc{var})$$

$$\frac{\Gamma, x : A\ \rhd\ M : B}{\Gamma\ \rhd\ \lambda x{:}A.\, M : \Pi x{:}A.\, B} \quad (\lambda)$$

$$\frac{\Gamma\ \rhd\ M : \Pi x{:}A.\, B \qquad \Gamma\ \rhd\ N : A}{\Gamma\ \rhd\ M\, N : B[N/x]} \quad (\textsc{app})$$

$$\frac{\Gamma\ \rhd\ M : A \qquad \Gamma\ \rhd\ A : \mathsf{Power}(B)}{\Gamma\ \rhd\ M : B} \quad (\textsc{sub})$$

$$\frac{\Gamma\ \rhd\ M : \Pi \vec{x}{:}\vec{A}.\ \mathsf{Power}(B)}{\Gamma\ \rhd\ M : \Pi \vec{x}{:}\vec{A}.\ \mathsf{Power}(M\,\vec{x})} \quad (\textsc{refl})$$

$$\frac{\Gamma\ \rhd\ M : A \qquad \Gamma\ \rhd\ A = B : \mathsf{Power}(C)}{\Gamma\ \rhd\ M : B} \quad (\textsc{conv})$$

$$\frac{\Gamma\ \rhd\ A' : \mathsf{Power}(A) \qquad \begin{array}{c}\Gamma, x : A'\ \rhd\ B : \mathsf{Power}(B') \\ \Gamma, x : A\ \rhd\ B : \mathsf{Power}(C)\end{array}}{\Gamma\ \rhd\ \Pi x{:}A.\, B : \mathsf{Power}(\Pi x{:}A'.\, B')} \quad (\Pi)$$

$$\frac{\Gamma\ \rhd\ A : \mathsf{Power}(B)}{\Gamma\ \rhd\ \mathsf{Power}(A) : \mathsf{Power}(\mathsf{Power}(B))} \quad (\mathsf{Power})$$

Figure 1: Typing for λ_{Power}

$$\frac{}{\rhd \langle\rangle} \quad \text{(EMPTY)}$$

$$\frac{\rhd \Gamma \qquad \Gamma \rhd A : \mathit{Power}(B)}{\rhd \Gamma, x : A} \quad \text{(EXTEND)}$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd M = M : A} \quad \text{(EQ-REFL)}$$

$$\frac{\Gamma \rhd N = M : A}{\Gamma \rhd M = N : A} \quad \text{(EQ-SYM)}$$

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd N = P : A}{\Gamma \rhd M = P : A} \quad \text{(EQ-TRANS)}$$

$$\frac{\Gamma, x : A \rhd M = M' : B}{\Gamma \rhd \lambda x{:}A.\, M = \lambda x{:}A.\, M' : \Pi x{:}A.\, B} \quad \text{(EQ-}\lambda\text{)}$$

$$\frac{\Gamma \rhd M = M' : \Pi x{:}A.\, B \qquad \Gamma \rhd N = N' : A}{\Gamma \rhd M\, N = M'\, N' : B[N/x]} \quad \text{(EQ-APP)}$$

$$\frac{\Gamma, x : A \rhd M : B \qquad \Gamma \rhd N : A}{\Gamma \rhd (\lambda x{:}A.\, M)\, N = M[N/x] : B[N/x]} \quad \text{(EQ-}\beta\text{)}$$

$$\frac{\Gamma \rhd M : \Pi x{:}A.\, B}{\Gamma \rhd \lambda x{:}A.\, M\, x = M : \Pi x{:}A.\, B} \quad \text{(EQ-}\eta\text{)}$$

Figure 2: Context formation and equality for λ_{Power}

Finally, (*Power*) is the rule of power typing which allows iteration of the *Power* constructor: intuitively, if A is a subset of B, then the collection of all subsets of A is a subset of the collection of all subsets of B.

Equality rules (Figure 2.)
These rules are standard. With subtyping, they lead to several more rules, which don't need to be added to the system specially. These are shown in Proposition 4.9 and Proposition 4.10. In the rule (EQ-η), the usual side condition $x \notin FV(M)$ is intended.

Now we can be more precise about terms, types and subtypes.

Definition 3.1 (Terms, types and subtypes).
1. M is a Γ*-term* if for some A, $\Gamma \rhd M : A$,
2. A is a Γ*-type* if for some B, $\Gamma \rhd A : \mathsf{Power}(B)$, and
3. A is a subtype *of* B *in* Γ if $\Gamma \rhd A : \mathsf{Power}(B)$.

The adjective "well-formed" emphasises that a pre-term can be typed in the calculus, as required by Definition 3.1. (I say "well-formed" rather than "well-typed" to avoid confusion when another kind of type enters the picture in Section 5.) There are three derived judgement forms:

$$\begin{array}{lll} \Gamma \rhd A \le B & =_{\text{def}} & \Gamma \rhd A : \mathsf{Power}(B) \\ \Gamma \rhd A\ \mathsf{type} & =_{\text{def}} & \text{for some } B, \quad \Gamma \rhd A : \mathsf{Power}(B) \\ \Gamma \rhd A = B & =_{\text{def}} & \text{for some } C, \quad \Gamma \rhd A = B : \mathsf{Power}(C) \end{array}$$

Section 4 shows that these definitions make sense.

In this paper I will be less aggressive than I could be with the derived judgements above (and with the ones for bounded quantification and abstraction shown in Section 1); this is to be sure that the presence of power types is not forgotten!

4 Properties of λ_{Power}

In this section I shall establish some meta-properties of the type system λ_{Power}.

Since the three judgement forms are defined simultaneously, several meta-properties are proved by simultaneous induction on the number of rules in a derivation of an arbitrary judgement. Meta-properties which are proved in this way are often expressed as *admissible rules*. By contrast, *derivable rules* are those which can be constructed simply by a composition of rules of the system.

Proposition 4.1 (Derivable rules).
These rules are derivable in λ_{Power}*:*

$$\frac{\Gamma \rhd A\ \mathsf{type}}{\Gamma \rhd A \le A} \qquad \text{(SUB-REFL)}$$

$$\frac{\Gamma \rhd A \leq B \qquad \Gamma \rhd B \leq C}{\Gamma \rhd A \leq C} \tag{SUB-TRANS}$$

$$\frac{\Gamma \rhd A \ \mathit{type}}{\Gamma \rhd A = A} \tag{EQT-REFL}$$

$$\frac{\Gamma \rhd B = A}{\Gamma \rhd A = B} \tag{EQT-SYM}$$

Proof The rule (SUB-REFL) is just an instance of (REFL). The rule (SUB-TRANS) follows from (SUB) and (*Power*). The rules (EQT-REFL) and (EQT-SYM) are just special cases of (EQ-REFL) and (EQ-SYM) respectively. □

I distinguish derivable rules from those which are admissible but not derivable because in the semantics we consider some important admissible rules (namely, substitution and thinning) as part of the system, making sure they are valid in every model. Some authors add these "important" admissible rules to the presentation but this spoils the inductive proof of several meta-properties. So I stick with a presentation in which they are easily proved admissible.

Proposition 4.2 (Church-Rosser property (CR)).
The β-reduction relation on pre-terms of λ_{Power} satisfies the Church-Rosser property.

Proof By a straightforward extension of the Tait–Martin-Löf proof, for example [23]. □

Recall that no variable is declared more than once in a pre-context.

Notation 4.3. Let $\Gamma \equiv x_1 : A_1, \ldots$ be a pre-context.

- Let $\mathit{Dom}(\Gamma) =_{\text{def}} \{x_1, \ldots\}$ be the set of variables Γ declares,
- $\Gamma|_{x_i} =_{\text{def}} x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$ be the restriction of Γ up to x_{i-1}.
- Define $\Gamma(x_i) =_{\text{def}} A_i$, viewing Γ as a partial mapping $\Gamma : \boldsymbol{V} \rightharpoonup \boldsymbol{T}$.
- Define $\Gamma \subseteq \Gamma'$ iff every declaration $x_i : A_i$ in Γ also appears in Γ'.

I use J to range over judgements of the system, and $\Gamma \rhd J$ for a judgement with context Γ. One derivation is "shorter" than another if it has fewer rule instances in its proof tree; this measure is used in many proofs.

A *simultaneous substitution* is a partial map from variables to pre-terms; a *renaming* is the special case of a simultaneous substitution which is a bijection on a subset of $\boldsymbol{V}$. Substitution is extended to contexts and judgements componentwise, *e.g.,* if $\Gamma \equiv x_1 : A_1, \ldots$ then $\Gamma[N/x] \equiv x_1 : A_1[N/x], x_2 :, \ldots$.

Proposition 4.4 (Contexts).
Let $\Gamma \equiv x_1 : A_1, \ldots x_n : A_n$ be a pre-context. Then:

1. (Formation).
   If $\rhd \Gamma$ then there are shorter derivations of $\Gamma|_{x_i} \rhd A_i$ type for each $1 \leq i \leq n$.

2. (Correctness).
If $\Gamma \rhd J$ where J is a typing or equality judgement, then $\rhd \Gamma$ with a shorter derivation, and $FV(J) \subseteq Dom(\Gamma)$.

3. (Renaming).
If $\Gamma \rhd J$ and Φ is a renaming of $Dom(\Gamma)$, then $\Phi(\Gamma) \rhd \Phi(J)$.

4. (Thinning).
If $\Gamma \rhd J$ and $\Gamma \subseteq \Gamma'$ with $\rhd \Gamma'$, then $\Gamma' \rhd J$.

Proposition 4.5 (Substitution).
$\Gamma, x : A, \Gamma' \rhd J \quad and \quad \Gamma \rhd M : A \quad \Longrightarrow \quad \Gamma, \Gamma'[M/x] \rhd J[M/x]$.

Proposition 4.6 (Bound narrowing).
$\Gamma, x : A, \Gamma' \rhd J \quad and \quad \Gamma \rhd A' : \mathsf{Power}(A) \quad \Longrightarrow \quad \Gamma, x : A', \Gamma' \rhd J$.

Proof Follows from Propositions 4.4 and 4.5, and the rule (SUB). In detail: let y be a fresh variable. Then using Proposition 4.4 (Proposition 2, then Proposition 1 repeatedly together with Proposition 4) and the assumption we can derive $\Gamma, y : A', x : A, \Gamma' \rhd J$. By the variable rule and subsumption, $\Gamma, y : A' \rhd y : A$. Hence by Proposition 4.5, $\Gamma, y : A', \Gamma'[y/x] \rhd J[y/x]$. The result follows by renaming y back to x using Proposition 4.4(3). □

The next proposition establishes simple formation properties of typing compound terms.

Proposition 4.7 (Formation).
1. $\Gamma \rhd \lambda x{:}A.\, M : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathsf{type} \quad and \quad \exists B.\ \ \Gamma, x : A \rhd M : B$.

2. $\Gamma \rhd M\,N : C \quad \Longrightarrow \quad \exists A, B.\ \ \Gamma \rhd M : \Pi x{:}A.\, B \quad and \quad \Gamma \rhd N : A$.

3. $\Gamma \rhd \Pi x{:}A.\, B : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathsf{type} \quad and \quad \Gamma, x : A \rhd B\ \mathsf{type}$.

4. $\Gamma \rhd \mathsf{Power}(A) : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathsf{type}$.

Moreover, for each part, the judgements in the consequence occur as subderivations of the judgement in the antecedent.

Proof By induction on derivations. In each case, following the left branch of the tree we eventually meet the rule that introduced the constructor, either (λ), (APP), (Π) or (*Power*). The premises of this rule give the result required. □

The next proposition states that every pre-term that appears in the predicate position of a judgement (*i.e.,* on the right of ':') is indeed what we call a type.

Proposition 4.8 (Type correctness).
1. $\Gamma \rhd M : A \quad \Longrightarrow \quad \Gamma \rhd A\ \mathsf{type}$.

2. $\Gamma \rhd M = N : A \quad \Longrightarrow \quad \Gamma \rhd A\ \mathsf{type} \quad and \quad \Gamma \rhd M, N : A$.

Proof The parts are proved together, by induction on derivations, using Proposition 4.7 in several cases, also Propositions 4.4 and 4.5.

Case (ATOMIC): By (*Power*).

Case (VAR): Using Proposition 4.4.

Case (λ): By the induction hypothesis, Proposition 4.4 and (Π).

Case (APP): By the induction hypothesis and Proposition 4.7 for the first premise, then Proposition 4.5 with the second premise.

Case (SUB): By the induction hypothesis and Proposition 4.7.

Case (EQ): Immediate by the second premise.

Case (*Power*): By the induction hypothesis and (*Power*) again.

Case (Π): Suppose $A \equiv \Pi x{:}C'.\,D'$. Then by (SUB-REFL) and the second premise we have $\Gamma \rhd C' : \mathsf{Power}(C')$. By the induction hypothesis for the third premise and Proposition 4.7, we have $\Gamma,\, x : C' \rhd D'\ \mathsf{type}$. Using ($\Pi$) again we can derive $\Gamma \rhd \Pi x{:}C'.\,D'\ \mathsf{type}$, and the result follows by (*Power*). □

The few basic equality rules of Figure 2 have some important admissible rules as consequences, proved using the propositions above. These include congruence rules for the type constructors, and rules of subsumption, conversion and substitution for the equality judgement itself.

Proposition 4.9 (Admissible equality rules).

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A = B}{\Gamma \rhd M = N : B} \tag{EQ-CONV}$$

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd M = N : B} \tag{EQ-SUB}$$

$$\frac{\Gamma,\, x : A \rhd M = M' : B \qquad \Gamma \rhd N = N' : A}{\Gamma \rhd M[N/x] = M'[N'/x] : B[N'/x]} \tag{EQ-SUBST}$$

$$\frac{\begin{array}{ll}\Gamma \rhd A' \leq A & \Gamma,\, x : A' \rhd B \leq B' \\ \Gamma,\, x : A \rhd M : B & \Gamma,\, x : A' \rhd M = M' : B'\end{array}}{\Gamma \rhd \lambda x{:}A.\, M = \lambda x{:}A'.\, M' : \Pi x{:}A'.\, B'} \tag{EQ-λ'}$$

Proof Each case is derived using lambda-abstraction, application, beta-conversion, together with symmetry, transitivity and Proposition 4.8. We omit details and just mention the other rules used. For the rule (EQ-SUB), we use (SUB); for the rules (EQ-CONV) and (EQ-SUBST) we use (CONV). For (EQ-λ'), there is an essential use of (EQ-η). □

The rule (EQ-λ') is a more general rule of λ-equality which allows us to restrict the domain of comparison of two functions.

Proposition 4.10 (Admissible type equality rules).

$$\frac{\Gamma \rhd A = B \qquad \Gamma \rhd B = C}{\Gamma \rhd A = C} \tag{EQT-TRANS}$$

$$\frac{\Gamma \rhd A = B}{\Gamma \rhd \mathsf{Power}(A) = \mathsf{Power}(B)} \tag{EQ-\textit{Power}}$$

$$\frac{\Gamma, x : A \rhd B = B'}{\Gamma \rhd \Pi x{:}A.\, B = \Pi x{:}A.\, B'} \tag{EQ-Π}$$

Proof The congruence rules (EQ-*Power*) and (EQ-Π) are proved as in Proposition 4.9, using beta-conversion. The transitivity rule (EQT-TRANS) follows from the admissibility of another rule:

$$\frac{\Gamma \rhd A = B : \mathsf{Power}(C)}{\Gamma \rhd A = B : \mathsf{Power}(B)} \tag{EQ-SUB-REFL}$$

since from the first premise of (EQT-TRANS) we can then show $\Gamma \rhd A = B : \mathsf{Power}(B)$ and by symmetry twice, $\Gamma \rhd B = C : \mathsf{Power}(B)$, and finally $\Gamma \rhd A = C$ using (EQ-TRANS). We can show that the rule (EQ-SUB-REFL) is admissible using similar reasoning as for the congruence rules, using (CONV). □

The rule (EQ-SUB-REFL) shows that type equality is "absolute", in the sense that the derivability of $A = B : \mathsf{Power}(C)$ is not affected by the choice of C when A and B are types such that $A, B : \mathsf{Power}(C)$.[6] In general, we expect this for type equality, but not necessarily for term equality. It is typical for subtyping calculi that the equality of two terms may vary across their common types. The semantics considered later reflects these ideas.

4.1 Further properties

We would like to prove more about the λ_{Power} system than the properties in the previous section. One desirable property is the important practical property of subject reduction:

If $\Gamma \rhd M : A$ and $M \twoheadrightarrow_{\beta\eta} M'$, then $\Gamma \rhd M' : A$ too.

Subject reduction allows type-checking algorithms to use untyped reduction to simplify terms, and for a compiled language based on the type system, it justifies the removal of run-time type-checking.

Unfortunately it seems difficult to prove for λ_{Power}. Several other desirable properties are closely related to subject reduction and may be as hard to prove. This section discusses the problems in attempting to prove subject reduction, and possible solutions.

[6] If there is a C' such that $B : \mathsf{Power}(C')$ then $A = B : \mathsf{Power}(C')$ too by (EQ-SUB-REFL), (*Power*), and subsumption for equality.

The key to proving subject reduction is to first establish a *generation principle*, which gives a way of decomposing derivations by stating how a particular judgement was derived. A generation principle is important for meta-theoretic analysis, and leads to other results besides subject reduction. In deterministic type-systems where there is a correspondence between typing rules and term constructors and every term has a unique type, a generation principle is trivial to derive, amounting to an inversion of the rules. In a system with dependent types or subtyping, the statement and proof of generation principles is made difficult because the presentation is not syntax-directed — so the last rule of a derived judgement is not uniquely determined by its form.

Proposition 4.7 is a weak generation principle, but it is not strong enough. For a judgement $\Gamma \rhd N : C$, we need a principle which connects C with the judgements about subterms of N asserted to exist. For the λ-case, a first approximation might be this:

> $\Gamma \rhd \lambda x{:}A.\, M : C \quad \Longrightarrow \quad C = \Pi x{:}A'.\, B'$
> where $\Gamma \rhd A' \leq A$ and there is a B such that $\Gamma,\, x : A \rhd M : B$ and $\Gamma,\, x : A' \rhd B \leq B'$.

This captures the observation that after applying (λ)to derive $\lambda x{:}A.\, M : \Pi x{:}A.\, B$, there can be several subsumptions and conversions through which $\Pi x{:}A.\, B$ mutates into C:

$$\cfrac{\cfrac{\cfrac{\cfrac{x : A \rhd M : B}{\lambda x{:}A.\, M : \Pi x{:}A.\, B}}{\vdots}}{\lambda x{:}A.\, M : C_j \qquad C_j \leq C_{j+1}}}{\lambda x{:}A.\, M : C_{j+1}}\ (\textsc{sub})$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\lambda x{:}A.\, M : C_{j+1}}{\vdots}}{\lambda x{:}A.\, M : C_k \qquad C_k = C_{k+1}}}{\lambda x{:}A.\, M : C_{k+1}}\ (\textsc{conv})}{\vdots}$$

$$\lambda x{:}A.\, M : C$$

The cut-like rules (SUB) and (CONV) make it hard to prove the statement directly, because to "join up" the arbitrary C_i's in the intervening typings we nant to use the generation principle being proved. It is worse than this, because (REFL) can introduce other detours, so the putative statement above needs altering to allow for the case that $C = \Pi x{:}A.\ \mathsf{Power}(Nx)$.

The traditional syntactic solution to this problem is to give a *syntax-directed* reformulation of the system, eliminating the cut-like rules. The generation principle for the cut-free system is immediate. This programme has been carried out for several subtyping systems by now [15, 13], including the first dependent type system $\lambda P_{\leq}$ by Aspinall and Compagnoni [3]. Unfortunately this technique does not apply easily to λ_{Power}. The sticking point is bounded operator abstraction which makes it hard to prove substitution lemmas in the syntax-directed system before proving other properties which depend on substitution. For $\lambda P_{\leq}$, we managed to find a judicious order in which to prove these properties.

A related solution involves giving a revised definition of the subtyping relation from the outset, on pre-terms. This too is difficult for power types, which have no separate subtyping judgement anyway. The problem remains open.

A new proof technique has been developed recently by [12] may help. Several desirable properties are proved at once using a *typed operational semantics* for subtyping, which is a deterministic reformulation of the system with extensive type annotations and control

over reduction order. The cost is having to prove equivalence between the typed operational semantics and the original presentation using a model-theoretic construction. But in future work, perhaps a generalization of the model definition for λ_{Power} given in Section 6 could be used to apply this new technique.

Summary of further properties

This is a summary of further properties we would like to prove for λ_{Power}.

Conjecture 4.11 (Strengthening).
$\Gamma, x : A, \Gamma' \rhd J \quad and \quad x \notin FV(\Gamma', J) \quad \Longrightarrow \quad \Gamma, \Gamma' \rhd J.$

Conjecture 4.12 (Variable Generation).
$\Gamma, x : A \rhd x : B \quad and \quad x \notin FV(B) \quad \Longrightarrow \quad \Gamma \rhd A : \mathsf{Power}(B).$

Conjecture 4.13 (Closure under reduction).
$\Gamma \rhd J \quad and \quad J \twoheadrightarrow_{\beta\eta} J' \quad \Longrightarrow \quad \Gamma \rhd J'.$

The well-known proofs of these properties are connected. To show closure for η-reduction and variable generation we need strengthening; to show closure we need a generation principle which has the variable generation result as a special case.

4.2 Remarks about the presentation of λ_{Power}

In this section I shall discuss a few aspects of the presentation of λ_{Power}, comparing with other type theories. This is partly to justify the definition, which has several variations from traditional type theory (as well as variations from traditional subtyping systems).

Lack of type-formation judgement

Typically, dependent type theories have a judgement form

$$\Gamma \rhd A \ \mathsf{type}$$

which asserts that A is a well-formed type. Philosophically, this should be a more primitive judgement than "$\Gamma \rhd M : A$" which asserts that the term M has type A. Indeed, it is conventional in Martin-Löf style presentations to assume that any rule with the conclusion $\Gamma \rhd M : A$ has $\Gamma \rhd A \ \mathsf{type}$ as a tacit premise.

In λ_{Power}, typing judgements of the form $\Gamma \rhd A : \mathsf{Power}(B)$ assert that A is a type, so to find out whether A is a type we must find a supertype of it. One supertype is A itself; $A : \mathsf{Power}(A)$ holds for the canonical types (non-variables/applications) because atomic types are supertypes of themselves by (ATOMIC) and the rules (Π), (Power) decompose Π and Power types. Non-canonical types (variables and applications) also inhabit Power-types, although showing this may take several steps of conversion and subsumption.

It turns out to be much easier to present the syntax of λ_{Power} without a primitive type-formation judgement than to attempt a formulation which separates types from terms

more fully. To separate types from terms we must either duplicate the rules for abstraction and application, or we must have a rule:

$$\frac{A : \mathit{Power}(B)}{A\ \mathit{type}}$$

which collapses the two notions anyway.

Having said that it would be more complex, a presentation with a type-formation judgement might still be useful, because it would allow type formation to be interpreted in a different way to typing in the semantics.

Multiple-identity

In Martin-Löf type theory, a well-formed string is either a term or a type. In λ_{Power}, we allow "confusion" between the notions, so that a pre-term can be a term or a type. [27] calls this confusion "multiple-identity". It is a throw back from classical set theory, where everything is a set. The original presentations of the Calculus of Constructions and the Extended Calculus of Constructions [for references, see 27] have multiple-identity, but in many recent studies of both syntax and semantics, it is eschewed. One reason for this is because types and terms are interpreted differently, so some way of disambiguating is needed to define the interpretation function.

To present λ_{Power} in a way which avoids multiple-identity, we could treat the elements of $\mathit{Power}(A)$ as *names* of types (*à la* Tarski), introducing rules:

$$\frac{M : \mathit{Power}(A)}{\mathit{El}(M)\ \mathit{type}} \qquad \frac{M : \mathit{Power}(A)}{\mathit{El}(M) \leq A}$$

Studying this idea might lead to a more type-theoretic explanation of power types; but it would enlarge the presentation considerably, since we would need rules for manipulating such types, and proving their equality.

Levels of power types

Many typed λ-calculi can be split into a finite number of "levels"; intuitively the level of a non-functional term is the number of colons that may appear before it. (This can be made into a definition which extends to functions, see [5].) If $M : A$ is derivable, then $\mathit{level}(A) = \mathit{level}(M) + 1$. Commonly, terms of level 0 are called "terms", terms of level 1 are called "types" and terms of level 2 are called "kinds".

Because of (ATOMIC) and (*Power*), λ_{Power} has an unbounded number of levels. In the empty context we can derive:

$$\rhd \kappa : \mathit{Power}(\kappa) : \mathit{Power}(\mathit{Power}(\kappa)) : \cdots \mathit{Power}^n(\kappa) : \cdots$$

Limiting the nesting of power types leads to natural sub-languages of λ_{Power} which have a finite number of levels. Such languages may be interesting in proof techniques or model constructions; there is a similarity to the Extended Calculus of Constructions with an unbounded hierarchy of universes [27].

How deeply nested do power types get? In Section 2.3 there were terms like $\mathit{Power}(\mathtt{Sort})$ which have level 3; to be able to form such a thing we need to have terms with level 4. I don't know any natural examples of terms which need a deeper nesting than this.

Semantic versus syntactic presentation

The presentation of λ_{Power} given in Figure 1 and 2 uses an equality judgement. This form of presentation for dependent type systems is sometimes described as *semantic*, in contrast to a presentation which defines a rule of conversion using untyped conversion, described as a *syntactic* presentation.

$$\frac{\Gamma \rhd M : A \qquad \Gamma \rhd A = B}{M : B} \qquad \frac{\Gamma \rhd M : A \qquad A =_{\beta\eta} B \qquad \Gamma \rhd B \ \mathit{type}}{M : B}$$

The semantic presentation, using the first rule, is needed to give a proper proof of soundness for dependent types. At once it excludes the possibility that untypable terms can occur in typing derivations, which the second rule does not.

If the system has the Church-Rosser property and subject reduction, it follows that only typable terms need occur in derivations in a syntactic presentation [19]. The metatheory of syntactic presentations seems to be more tractable in the sense that there may be combinatorial proofs of properties such as subject reduction (although once η-reduction is included, these proofs rely on strong normalization [17]). For semantic presentations, the typical method of attack has been via a model construction [37, 20].

Pure Type Systems [5] are defined with a syntactic presentation using the second rule above. A variation on this is to use a split rule of conversion: a rule of type-reduction and a rule of well-formed type-expansion. With this scheme, the Church-Rosser property is not needed to argue that untypable terms do not occur in derivations, but subject reduction remains essential. Unfortunately the sketched proof of the subject reduction property by Sannella, Sokołowski, and Tarlecki was incomplete and seems difficult to fix, as explained in Section 4.1.

Rules of η Equality

A theory of $\beta\eta$ conversion generated by the compatible closure of $\beta-$ and $\eta-$ reduction identifies terms which differ only in type-labels on λ-abstractions. This can be seen by the usual "failure of confluence" example:

$$\lambda y{:}B.\, M\eta_{-} - \lambda x{:}A.\, (\lambda y{:}B.\, M)\, x \longrightarrow_{\beta} \lambda x{:}A.\, M$$

(It is well-known that the combination of $\beta\eta$ reduction is confluent on untyped lambda terms).

When $\beta\eta$ conversion is restricted to terms of the same type, this failure disappears in type systems without subtyping; we have that if $\lambda x{:}A.\, M : C$ and $\lambda x{:}B.\, M : C$, then $\lambda x{:}A.\, M =_{\beta\eta} \lambda x{:}B.\, M$ implies $A =_{\beta\eta} B$. This fairly easy to show for simply-typed $\lambda^{\to}$ [30], and was shown for Pure Type Systems by Geuvers [17].

This is exactly how (EQ-λ') in Proposition 4.9 is derived in the typed equational theory. In the typed setting, the equality of terms is restricted to those which can be typed with the same Π-type, and moreover, for which $A \leq B$.

5 Rough Type-checking

Although λ_{Power} is a dependently-typed calculus, we can approximate type-checking using "rough" types without term dependency. Rough type-checking is useful because it

$$\frac{}{\blacktriangleright \langle\rangle}$$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma \blacktriangleright A : \mathsf{P}(\tau)}{\blacktriangleright \Gamma, x : A}$$

$$\frac{\blacktriangleright \Gamma}{\Gamma \blacktriangleright \kappa : \mathsf{P}(\kappa)}$$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma|_x \blacktriangleright \Gamma(x) : \mathsf{P}(\tau)}{\Gamma \blacktriangleright x : \tau}$$

$$\frac{\Gamma \blacktriangleright A : \mathsf{P}(\tau) \qquad \Gamma, x : A \blacktriangleright M : \upsilon}{\Gamma \blacktriangleright \lambda x{:}A.M : \tau \Rightarrow \upsilon}$$

$$\frac{\Gamma \blacktriangleright M : \tau \Rightarrow \upsilon \qquad \Gamma \blacktriangleright N : \tau}{\Gamma \blacktriangleright M\,N : \upsilon}$$

$$\frac{\Gamma \blacktriangleright A : \mathsf{P}(\tau) \qquad \Gamma, x : A \blacktriangleright B : \mathsf{P}(\upsilon)}{\Gamma \blacktriangleright \Pi x{:}A.B : \mathsf{P}(\tau \Rightarrow \upsilon)}$$

$$\frac{\Gamma \blacktriangleright A : \mathsf{P}(\tau)}{\Gamma \blacktriangleright \mathsf{Power}(A) : \mathsf{P}(\mathsf{P}(\tau))}$$

Figure 3: Rough typing for λ_{Power}

enforces a structural well-formedness property that is necessary for typability in the full system. Two pre-terms which are in the full typing relation of λ_{Power} have related rough types, and two terms which are equal in the equational theory have the same rough type. It is easy to check rough typability because we don't need to test equality of terms, so we have *static* type-checking.

The idea of rough type-checking comes from [34][7], which suggested that rough types could be used to give a foundationally-secure semantics to ASL+. This is done for λ_{Power} in Section 6. Another application of rough types is the proof of strong normalization for λ_{Power} . This proof is given in Section 8.

5.1 Rough typing system

Given a set $\mathcal{K}$ of atomic types, the set $\boldsymbol{Ty}_{\mathcal{K}}$ of *rough types over* $\mathcal{K}$ consists of type constants, arrow types, and power types, defined by the grammar:

$$\boldsymbol{Ty} \quad ::= \quad \mathcal{K} \quad | \quad \boldsymbol{Ty} \Rightarrow \boldsymbol{Ty} \quad | \quad \mathsf{P}(\boldsymbol{Ty})$$

(writing $\boldsymbol{Ty}$ as short for $\boldsymbol{Ty}_{\mathcal{K}}$). These are the types of the simply-typed λ-calculus extended with a power type constructor. I use $\tau, \upsilon, \ldots$ to range over $\boldsymbol{Ty}$.

There are two rough typing judgements, using filled triangles:

$\blacktriangleright\ \Gamma$	Γ is a roughly-typable context
$\Gamma \blacktriangleright M : \tau$	M has rough type τ in Γ

The judgements are defined inductively by the rules in Figure 3. Notice that full λ_{Power} contexts are used in the rough typing judgements; at the cost of extra judgement forms and syntax we could instead introduce rough contexts, declaring variables with rough types, and then type-check a λ_{Power} context to a rough context. At the moment this is done "on-the-fly" by the variable rule, but the idea of a rough context is behind the scenes (and would be used in a practical implementation).

One can understand the rough typing rules as an abstract interpretation of terms-in-context, which follows set-theoretic intuitions for the calculus. The rough type of a term tells us what kind of beast it denotes: lambda terms denote functions and have arrow rough-types; atomic types and power types denote collections of values and have power rough-types. A term $\Pi x{:}A.B$ has a rough type of the form $\mathsf{P}(\tau \Rightarrow \upsilon)$, indicating that it denotes a collection of functions.

Example 5.1. To illustrate rough typing, recall the example context Γ_{PERM} from Section 2.1. We can derive these rough typings:

$$\Gamma_{\mathrm{PERM}} \blacktriangleright \mathsf{Perm} : \mathsf{int} \Rightarrow \mathsf{P}(\mathsf{int} \Rightarrow \mathsf{int})$$

$$\Gamma_{\mathrm{PERM}} \blacktriangleright \mathsf{Invperm} : \mathsf{int} \Rightarrow (\mathsf{int} \Rightarrow \mathsf{int}) \Rightarrow (\mathsf{int} \Rightarrow \mathsf{int}).$$

At once we see how "rough" this is: *Perm* and *Invperm* were defined on *nat*, but *nat* gets replaced by the atomic type *int*.

In general, rough typing judgements — or to be more precise, their translation got by mapping $\tau \Rightarrow \upsilon$ to $\Pi x{:}\tau.\upsilon$ — do not hold in the full λ_{Power} type system. Certainly we do *not* have:

$$\Gamma_{\mathrm{PERM}} \triangleright \mathsf{Perm} \ : \ \mathsf{int} \to \mathsf{Power}(\mathsf{int} \to \mathsf{int})$$

because, for starters, *Perm* is not defined on all of *int*.

In Proposition 5.6, we prove that typability in the full calculus guarantees rough typability. The above example shows that the converse fails, since:

$$\Gamma_{\mathrm{PERM}}, i : \mathsf{int} \blacktriangleright \mathsf{Perm}\, i : \mathsf{P}(\mathsf{int} \Rightarrow \mathsf{int})$$

[7] Rough types were called "types" in [34], but using this name we couldn't call anything in the full system a "type". Rough types were called "kinds" in an early presentation of my work on ASL+ in [1], but this name confused some people. Another possibility is "simple type" which suggests the removal of dependency and relationship to $\lambda^{\to}$. I shall stick to "rough types" here because it gives extra intuition and relates to the explanation in [34] where a term is called "roughly well formed" if it has a rough type.

but $\mathsf{Perm}\ i$ cannot be typed in the full system[8].

Although rough typing judgements are generally not valid in the full system, given a context Γ and term M, it is possible to apply a "rounding up" operation which replaces all types appearing in variable declarations and bindings by terms corresponding to their rough types, so removing dependency. This translates any rough-typing derivation into a derivation in the full calculus. Comparing the rules in Figure 3 and Figure 1 it should be clear how to do this: the translated derivation uses only the typing rules of λ_{Power}, and not (SUB), (CONV) or (REFL). This gives a precise explanation of how the rough-typing system can be seen as a subsystem of the full system. More details of this "rounding up" operation are sketched by [35].

5.2 Properties of rough typing

It is easier to establish properties of the rough-typing system than the full system, because the types are non-dependent and subtyping has been removed.

Proposition 5.2 (Properties of rough typing).

1. *$\Gamma \blacktriangleright M : \tau$ implies $\blacktriangleright \Gamma$*
2. *If $\Gamma \blacktriangleright M : \tau$, then τ is the unique such rough type.*
3. *Rough type-checking and rough type-inference are decidable.*[9]

Proof

1. By a simple induction over derivations.
2. For a given M only one rough typing rule can have a matching conclusion.
3. For any rule in Figure 3, count the number of symbols to the left of the rough type in the conclusion. In each case, every premise of a rule has strictly fewer symbols, so we have a termination ordering for deciding the relation. Moreover, we can turn the rules into an algorithm for computing τ, since there is at most one rule which matches any M. □

Proposition 5.3 (Thinning and substitution for rough typing).

1. *If $\Gamma \blacktriangleright M : \tau$ and $\Gamma \subseteq \Gamma'$ with $\blacktriangleright \Gamma'$, then $\Gamma' \blacktriangleright M : \tau$.*
2. *If $\Gamma, x : A, \Gamma' \blacktriangleright M : \upsilon, \Gamma \blacktriangleright A : \mathsf{P}(\tau), \Gamma \blacktriangleright N : \tau$, then $\Gamma, \Gamma'[N/x] \blacktriangleright M[N/x] : \upsilon$.*

Proof Each part by induction on derivations.

1. Easy induction on the derivation.
2. By induction on the first derivation. We give the two interesting cases, others follow by the induction hypothesis.

   Case $M \equiv z$: If $z \equiv x$ then $\Gamma, \Gamma' \rhd N : \tau'$ by weakening, and $\tau' \leq \tau$ by assumption. If $z \not\equiv x$ then the result follows immediately if $z \in \mathit{Dom}(\Gamma)$, otherwise using the induction hypothesis.

[8] to prove this rigorously we need to use a generation principle or model construction.
[9] assuming we can decide syntactic identity of atomic types, *i.e.,* whether $\kappa \equiv \kappa'$.

Case $M \equiv M_1\, N_1$: where $\Gamma \rhd M_1 : \sigma \Rightarrow \tau$, $\Gamma \rhd N_1 : \sigma'$ and $\sigma' \leq \sigma$.
Then by induction hypothesis for the first premise,
$\Gamma, \Gamma'[N/x] \rhd M_1 : \sigma \Rightarrow_1 \tau_1$ with $\sigma \Rightarrow_1 \tau_1 \leq \sigma \Rightarrow \tau$.
By the definition of rough subtyping, $\sigma \leq \sigma_1$ and $\tau_1 \leq \tau$.
By the induction hypothesis for the second premise,
$\Gamma, \Gamma'[N/x] \rhd N_1 : \sigma_1'$ with $\sigma_1' \leq \sigma'$.
Now by transitivity of subtyping, $\sigma_1' \leq \sigma_1$, hence
$\Gamma, \Gamma'[N/x] \rhd M_1\, N_1 : \tau_1'$ by the application rule again. □

Proposition 5.4 (Strengthening for rough typing).
If $\Gamma, x : A, \Gamma' \blacktriangleright M : \tau$ and $x \notin FV(\Gamma'), FV(M)$, then $\Gamma, \Gamma' \blacktriangleright M : \tau$.

Proof By simultaneous induction also for the context rough-typing judgement, using free-variable properties of roughly-typable contexts (analogous to Proposition 2 of Proposition 4.4). □

The rough-typing system has a direct generation principle since there is a single typing rule for each term-former. We can show the subject reduction property without fuss.

Proposition 5.5 (Subject reduction for rough typing).
If $\Gamma \blacktriangleright M : \tau$ and $M \twoheadrightarrow_{\beta\eta} M'$, then $\Gamma \blacktriangleright M' : \tau$ too.

Proof Using generation, with substitution for β-reduction and with strengthening for η-reduction. □

The agreement property below is the important connection between rough types and typing in full λ_{Power}, claimed at the beginning of this section. Part 4 is just needed to make the induction go through.

Theorem 5.6 (Agreement of rough typing).

1. *If $\rhd \Gamma$ then $\blacktriangleright \Gamma$.*
2. *If $\Gamma \rhd M : A$ then for some $\tau \in \mathbf{Ty}$, $\Gamma \blacktriangleright M : \tau$ and $\Gamma \blacktriangleright A : \mathsf{P}(\tau)$.*
3. *If $\Gamma \rhd M = N : A$ then for some $\tau \in \mathbf{Ty}$, $\Gamma \blacktriangleright M, N : \tau$ and $\Gamma \blacktriangleright A : \mathsf{P}(\tau)$.*
4. *If $\Gamma \rhd A : \mathsf{Power}(B)$ then for some $\tau \in \mathbf{Ty}$, $\Gamma \blacktriangleright A : \mathsf{P}(\tau)$.*

Proof By induction on the derivation in the full system. □

Note that a consequence of this theorem is that if $\Gamma \rhd A : \mathsf{Power}(B)$ in λ_{Power}, then A and B are roughly typable, and must have the same rough type $\mathsf{P}(\tau)$ for some τ.

5.3 Remarks about rough typing

Type-checking in λ_{Power}. Rough types have a simpler structure than types in the full system — in particular, there is no λ-abstraction or application in rough types. This means that the rough type of a term may be *more* informative than a particular type in the full system. For example, from a judgement $\Gamma \rhd x : y\, z$, we need further analysis to tell whether x denotes a value of atomic type, a function, or a collection. Rough typing provides this analysis. It is important for type-checking algorithms, which have to determine whether something is a function or a type (which can be difficult with subtyping and bounded type variables).

Different atomic types. A generalization of the connection between λ_{Power} and its rough types would be to allow different sets of atomic types in each syntax, and consider a mapping from a type $\kappa \in \mathcal{K}$ in λ_{Power} to a type from a new set of atomic rough types. This would be useful for the semantics considered in the next section. However, it will be subsumed in $\lambda_{\mathsf{ASL+}}$ by other extensions.

λ_{Power} as a fragment of HOL. The rough-types of λ_{Power} can be seen as the types of a Church-style formulation of higher-order logic, taking $P(\tau)$ as an abbreviation for $\tau \Rightarrow \mathsf{Prop}$. Then a straightforward translation of the terms of λ_{Power} into HOL terms gives an understanding of λ_{Power} inside higher-order logic.

The idea for this is laid out in a general setting of dependent types by Jacobs and Melham [25]. To add the richer equational theory which results from the addition of subtyping, types would need to be interpreted as relations rather than predicates, so taking $P(\tau)$ as an abbreviation for $\tau \times \tau \Rightarrow \mathsf{Prop}$. This interpretation is closely connected with models considered in Section 6, and also with the understanding of subtyping as intuitionistic implication explained by Longo et al. [26].

Rough typing elsewhere. Rough type-checking makes sense for other type systems with dependent types, and may be a useful tool. It is similar in spirit to the reduction-preserving dependency-removal translations which are used to show strong normalization of the systems of the Lambda Cube, starting from Plotkin's original idea for LF [18, 22]. Strong normalization for λ_{Power} is proved in Section 8 using rough typing.

6 Semantics

In this section I shall give an environment model definition for λ_{Power} and some examples of models. This begins in Section 6.1. Before that I shall discuss some of the motivations for doing this, and how the model definition was arrived at.

Motivations

For constructive type theories, a semantical interpretation is often a secondary construction, perhaps a tool to prove consistency or strong normalization. The important thing for such theories is the syntax. But in the intended application of λ_{Power} to the specification language ASL+, there is an intended model where atomic types denote classes of algebras in some institution. Algebras are the denotations of programs, so a parameterised program denotes a mapping of algebras.

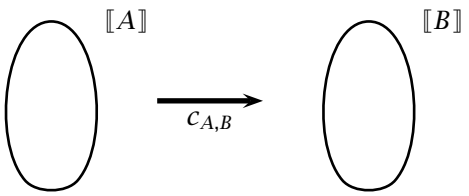
Starting with an intended model in mind, soundness of the interpretation is crucial, and also influences the model definition. Put crudely, the definition was arrived at by adjusting the set-theoretic model to handle the equational theory of λ_{Power}, and then abstracting far enough to capture a term model, sometimes a good tool for studying syntax. I do not claim to have abstractly characterised the structure of a λ_{Power} model, although it would be interesting to try to do that.

My adjustment of the set-theoretic model for ASL+ to capture the equational theory was first published in Aspinall [1], changing the model given in Sannella et al. [34] and

using rough typing. That model is an instance of the model definition given for $\lambda_{\mathsf{ASL+}}$ in the next chapter, which extends the model definition for λ_{Power} given here.

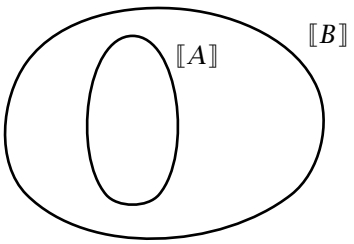
Containment versus coercion

Subtyping calculi have two basic kinds of model. With a typed value space, we may choose a *coercion semantics*, where each use of subsumption is modelled by the insertion of a *coercion* from type to supertype. If $A \leq B$, there is a map $c_{A,B} : [\![A]\!] \to [\![B]\!]$:



In some sense this is the most general setting, because coercions may be merely identities. But it requires a way of relating coercions to the syntax: either we forgo (SUB) and introduce coercions explicitly into the syntax [11], or we reconstruct coercions by a translation process [6]. Either route needs a *coherence* property of the interpretation, to show that different ways of putting coercions into a coercion-free judgement have the same interpretation. This coherence property can be quite difficult to establish in a general form, and has yet to be demonstrated in a subtyping calculus more complex than $F_{\leq}$. See [15] for this, or [31, Chapter 10] for the simpler example of $\lambda_{\leq}$.

The other kind of model is a *containment semantics* in which subtyping is interpreted as containment between types: if types are interpreted as sets, for example, then subtyping would be interpreted by subset inclusion:



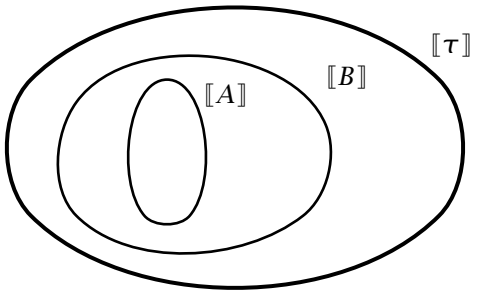
This may be appropriate when subtypes really are intended as subsets, which is the case for the ASL+. In programming languages, the coercion viewpoint may be more appropriate; there can be real work in coercing an integer into a real number, for example.

With a containment semantics there is no problem of coherence, but there is a fundamental difficulty with the contravariant rule for subtyping Π-types. In the syntax we have that $\mathit{int} \to \mathit{int} \leq \mathit{nat} \to \mathit{int}$, but this does not hold as a set-theoretic inclusion; $\mathbf{Z} \to \mathbf{Z} \not\subseteq \mathbf{N} \to \mathbf{Z}$ when the semantic $\to$ denotes set-theoretic function space. This is sometimes solved by interpreting $\mathit{nat} \to \mathit{int}$ as the collection of all partial functions defined *at least* on $\mathbf{N}$; then the inclusion $\mathbf{Z} \to \mathbf{Z} \subseteq \mathbf{N} \to \mathbf{Z}$ holds. Of course, we need a universe of values over which to form this "collection of all partial functions," and this is what leads to using an *untyped* value space in containment semantics.

Typically, the untyped value space in a containment semantics is the domain of a model of the untyped λ-calculus; a term is interpreted via the interpretation of its type-erasure [31]. But it is a surprising overkill to base a semantics for a calculus even as simple as $\lambda_{\leq}$, the extension of the simply-typed λ-calculus with subtyping, on a model of the untyped λ-calculus. For λ_{Power} I shall stay closer to the simpler set-theoretic semantics of the simply-typed λ-calculus.

A typed containment semantics for power types

I will give a containment semantics for λ_{Power} which is nevertheless based on a typed value space. Rough types make this possible. Whenever $A \leq B$ then A and B have the same rough type $\mathsf{P}(\tau)$, say, and so both may be interpreted as subsets of the interpretation of τ:



(in the models below, $[\![\tau]\!]$ is written as $\mathcal{D}^{\tau}$).

Since every type $\Pi x{:}C.\,D$ has a rough type of the form $\mathsf{P}(\tau_C \Rightarrow \tau_D)$, we can form the "collection of all functions with domain at least $[\![C]\!]$" using $[\![\tau_C]\!]$ as a universe, rather than a universal domain. Approximately,

$$[\![\Pi x{:}C.\,D]\!] = \left\{ f : [\![\tau_C]\!] \to [\![\tau_D]\!] \;\middle|\; \begin{array}{c} \forall c \in [\![C]\!],\ f(c) \text{ defined} \\ \text{and} \quad f(c) \in [\![D]\!]_{[x \mapsto c]} \end{array} \right\}$$

The main reason that for this form of interpretation is because of the intended model, where specification refinement is modelled as inclusion of classes of algebras. It only makes sense to consider algebras over the same *signature*; so signatures in the specification language ASL+ correspond to the atomic types of λ_{Power} here.

Another reason for the choice is that the set-theoretic intuition behind power types, that $\mathit{Power}(A)$ stands for the collection of *subsets* of A, fits better with a containment interpretation than with a coercion interpretation. A coercion interpretation interprets a separate subtyping judgement $A \leq B$ by constructing a map $c_{A,B} : [\![A]\!] \to [\![B]\!]$. Because we combine the typing and subtyping judgements it seems harder to do this. The type $\mathit{Power}(B)$ might be conceived as the "type of coercions into B" but then we need some way of recovering the domain of a coercion inhabiting $\mathit{Power}(B)$, and the details are messy.

From sets to PERs

There is another aspect to the structure of models. A *stratified equality* between terms is typical in a subtyping system: two terms may be equal considered at one type but not equal at a subtype. A model which interprets types as sets is not adequate for this. In the equation

$$\lambda x{:}A.\,M = \lambda x{:}A'.\,M' : \Pi x{:}A'.\,B'$$

the equality of the two sides depends on "restricting observations" to the type $\Pi x{:}A'.\,B'$. This leads to interpreting a type $A : \mathsf{P}(\tau)$ as a *partial equivalence relation* on $[\![\tau]\!]$; the partiality captures the subset part of the interpretation, and we have an equivalence relation because the equality judgement is an equivalence relation on terms at any one type.

(This use of PERs is *not* related to realizability: we use PERs over a hierarchy of domains rather than the single domain of a partial combinatory algebra.)

Because the full calculus is interpreted with respect to an interpretation of rough types, we have an *external equality* notion. In other words, rather than defining $[\![\lambda x{:}A.M : \Pi x{:}A'.\,B']\!]$ to be an equivalence class of values at type $\Pi x{:}A'.\,B'$, we give a fixed interpretation of $\lambda x{:}A.M$, depending only on its rough type. ***FIX ME!*** IS THIS TRUE? Doesn't depend only on rough type, does it?

Plan

The model definition for λ_{Power} is given in stages. First we give a definition of *applicative structure* which provides a semantic universe for the interpretation, together with fixed interpretations of the constants. This is in Section 6.1. Then we define notions of *environment* and *interpretation* for an applicative structure. An environment is used to interpret the free variables in a term. The interpretation itself is a mapping from well-typed terms and environments into the applicative structure. These definitions are in Section 6.2. Finally we give the model definition itself, in Section 6.3, which is a set of axioms that must be satisfied by an interpretation.

6.1 Structures

A λ_{Power} applicative structure is similar to a typed-applicative structure for $\lambda^{\to}$ (see *e.g.,* [30]) It provides semantic domains for every rough type; the domains are sets.

Definition 6.1 (λ_{Power} applicative structure).
A λ_{Power} *applicative structure* $\mathcal{D} = \langle \mathcal{D}, \mathsf{Const}, \mathsf{App} \rangle$ consists of

- a family of sets $\{\, \mathcal{D}^{\tau} \,\}_{\tau \in \mathbf{Ty}}$;
- a constant $\mathsf{Const}(\kappa) \in \mathcal{D}^{\mathsf{P}(\kappa)}$ for each $\kappa \in \mathcal{K}$, and
- a mapping $\mathsf{App}^{\tau,\upsilon} : \mathcal{D}^{\tau \Rightarrow \upsilon} \to \mathcal{D}^{\tau} \to \mathcal{D}^{\upsilon}$ for each $\tau, \upsilon \in \mathbf{Ty}$.

Type annotations τ, υ are sometimes omitted for brevity. □

The type-superscripts from the mappings $\mathsf{App}^{\tau,\upsilon}$ are usually omitted, but they will always be uniquely determined in formulae.

Examples of λ_{Power} structures are the full type hierarchy and the term structure. (In [2], a variant of the full type hierarchy with partial functions is also described).

Notation 6.2. Given a set S, $\mathrm{REL}(S)$ is the set of relations on S, $\mathrm{REL}(S) =_{\mathrm{def}} \mathit{Pow}(S \times S)$. If $R \in \mathrm{REL}(S)$, then $\mathit{dom}(R) = \{\, a \mid a\ R\ a \,\}$. A relation is a *partial equivalence* (PER) if it is symmetric and transitive; $\mathrm{PER}(S)$ is the set of PERs on S. The notation $a \mapsto f(a)$ stands for the function mapping a to $f(a)$, in other words, λ-abstraction in the meta-language.

Example 6.3 (Full hierarchy structure). Given a family of sets and PERs $C = \{\, C_\kappa, R_\kappa \in \mathrm{PER}(C_\kappa) \,\}_{\kappa \in \mathcal{K}}$, the *full hierarchy* $\mathcal{F}_C$ on C is defined by

$$
\begin{aligned}
\mathcal{F}^{\kappa} &= C_\kappa \\
\mathcal{F}^{\tau \Rightarrow \upsilon} &= \mathcal{F}^{\tau} \to \mathcal{F}^{\upsilon} \\
\mathcal{F}^{P(\tau)} &= \mathrm{REL}(\mathcal{F}^{\tau}) \\
\\
\mathit{App}(f, m) &= f(m) \\
\mathit{Const}(\kappa) &= R_\kappa
\end{aligned}
$$

□

In the full hierarchy structure, $\mathcal{F}^{P(\tau)}$ is the set of all relations over $\mathcal{F}^{\tau}$, rather than the set of all PERs. This is for a technical reason: because the interpretation in the full structure (Example 6.9) is defined over rough types, the type-constructors are not guaranteed to construct PERs. But PERs *are* guaranteed by the soundness result for the full calculus.

Using $\mathrm{REL}(\mathcal{F}^{\tau})$ is a design choice: instead we could adjust the interpretation (in Example 6.9) to return the empty relation — a PER trivially — in case of failure. The approach here seems less ugly, and opens the way towards models with different interpretations of types, models of reduction or program refinement, for example.

Example 6.4 (Full hierarchy partial function structure). Given a family of sets and PERs $C = \{ C_\kappa, R_\kappa \in \mathrm{PER}(C_\kappa) \}_{\kappa \in \mathcal{K}}$, the *full hierarchy partial function structure* $\mathcal{P}_C$ on C is defined by

$$
\begin{aligned}
\mathcal{P}^{\kappa} &= C_\kappa \cup \{ \diamond \} \\
\mathcal{P}^{\tau \Rightarrow \upsilon} &= (\mathcal{P}^{\tau} \rightharpoonup \mathcal{P}^{\upsilon}) \cup \{ \diamond \} \\
\mathcal{P}^{P(\tau)} &= \mathrm{REL}(\mathcal{P}^{\tau}) \cup \{ \diamond \} \\
\\
\mathit{App}(f, m) &= \begin{cases} \diamond & \text{if } f = \diamond,\ m = \diamond \text{ or } f(m) \text{ is undefined} \\ f(m) & \text{otherwise} \end{cases} \\
\mathit{Const}(\kappa) &= R_\kappa
\end{aligned}
$$

where $\diamond$ is a distinct token, not in any C_κ. □

Example 6.5 (Term structure). The *term structure* $\mathcal{T}_\Gamma$ of roughly-typed terms in Γ is given by

$$
\begin{aligned}
\mathcal{T}^{\tau} &= \{ M \mid \Gamma \blacktriangleright M : \tau \} \\
\mathit{App}(M, N) &= M\,N \\
\mathit{Const}(\kappa) &= \kappa
\end{aligned}
$$

□

6.2 Environments and interpretations

To define interpretations, we need a notion of environment. For each roughly-typable context Γ, we define a semantic domain $\mathcal{D}^{\Gamma}$ by induction on Γ, by

$$
\begin{aligned}
\mathcal{D}^{\langle\rangle} &= \{ \star \} \\
\mathcal{D}^{\Gamma, x:A} &= \mathcal{D}^{\Gamma} \times \mathcal{D}^{\tau} \qquad \text{where } \Gamma \blacktriangleright A : P(\tau)
\end{aligned}
$$

where $\{ \star \}$ is some singleton set. A *Γ-environment* is a nested tuple $\eta \in \mathcal{D}^{\Gamma}$. Because we use a name-free denotation, if Φ is a renaming on $\mathit{Dom}(\Gamma)$ then η is a $\Phi(\Gamma)$-environment iff it is a Γ-environment.

Given a Γ-environment $\eta \in \mathcal{D}^\Gamma$, we can define a projection function from the variables of Γ:

$$\begin{array}{rcll} \eta^{\langle\rangle}(y) & & \text{undefined, for all } y. \\ \eta^{\Gamma,x:A}(y) & = & \begin{cases} snd(\eta), & \text{if } y \equiv x, \\ (fst\,\eta)^\Gamma(y) & \text{if } y \not\equiv x. \end{cases} \end{array}$$

So if $\Gamma|_x \blacktriangleright \Gamma(x) : \mathsf{P}(\tau)$, then $\eta^\Gamma(x) \in \mathcal{D}^\tau$.

Thinning between environments is defined using this projection notation. If $\Gamma_1 \subseteq \Gamma_2$, $\eta_1 \in \mathcal{D}^{\Gamma_1}$ and $\eta_2 \in \mathcal{D}^{\Gamma_2}$, then ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$ iff

$$\eta_1^{\Gamma_1}(x) = \eta_2^{\Gamma_2}(x)$$

for all $x \in Dom(\Gamma_1)$.

The notation $\eta|_{x_i}$ stands for the restriction of a Γ-environment η to variables declared before x_i, meaning the shorter tuple $fst^{n-i}(\eta)$ where x_i is the ith variable of n declared in Γ.

Unlike a partial function environmentmapping variables to $\bigcup \{\, \mathcal{D}^\tau \,\}_{\tau \in \mathbf{Ty}}$, this tupled form has an explicit notion of the domain $\mathcal{D}^\Gamma$ associated to a context. We need this because relations over $\mathcal{D}^\Gamma$ are used in the soundness proof. Using tuples gives us an interpretation function reminiscent of the semantics of $\lambda^\to$ in (set-like) CCCs. Tuples are a rather concrete construction: more abstractly, contexts could be treated in the same way as types.

Definition 6.6 (λ_{Power} interpretation). A λ_{Power} *interpretation* in $\mathcal{D}$ consists of

- a meaning function $[\![\Gamma \blacktriangleright - : \tau]\!]_- : \mathbf{T} \to \mathcal{D}^\Gamma \to \mathcal{D}^\tau$, for each roughly-typable context Γ and $\tau \in \mathbf{Ty}$, such that whenever $\Gamma \blacktriangleright M : \tau$ and $\eta \in \mathcal{D}^\Gamma$, then $[\![\Gamma \blacktriangleright M : \tau]\!]_\eta \in \mathcal{D}^\tau$, and
- for each $\tau \in \mathbf{Ty}$, a mapping $\mathsf{Rel}^\tau : \mathcal{D}^{\mathsf{P}(\tau)} \to \mathrm{REL}(\mathcal{D}^\tau)$

Type annotations τ may be omitted for brevity. □

The meaning function in an interpretation is indexed by a rough type and a context; we use the judgement notation in $[\![\Gamma \blacktriangleright M : \tau]\!]_\eta$ just to indicate a roughly-typed term in context, not to indicate that the meaning function is defined on a derivation. (But it is often convenient to define meaning functions by induction over rough-typing derivations.) When Γ and τ are understood, we write just $[\![M]\!]_\eta$.

When $a \in \mathcal{D}^{\mathsf{P}(\tau)}$, I sometimes use R_a as shorthand for $\mathsf{Rel}^\tau(a)$. The mapping Rel models the behaviour (or *extension*) of elements denoting types, just as App models the extension of elements denoting functions. It is part of the interpretation so we can consider different "views" of types in the same structure.

Definition 6.6 does not require *a priori* that R_a is a PER, for the reason outlined before; instead the soundness theorem will imply that any type of λ_{Power} denotes a PER. This differs slightly from other model definitions for dependent types which use a *partial definition*, proved to be total on well-typed terms. Instead we require that an interpretation is defined on all *roughly-typed* terms.

We show how the structures of Examples 6.3–6.5 can be extended to interpretations in the next section.

6.3 Models

We will use some constructions on relations. Let $\mathcal{D}$ be a λ_{Power} interpretation. Given $R \in \mathrm{REL}(\mathcal{D}^\tau)$ and $G \in dom(R) \to \mathrm{REL}(\mathcal{D}^\upsilon)$, we define $\Pi(R,G) \in \mathrm{REL}(\mathcal{D}^{\tau\Rightarrow\upsilon})$, $\mathcal{P}wr(R) \in \mathrm{PER}(\mathcal{D}^{P(\tau)})$ by:

$$f \ \ \Pi(R,G) \ \ g \qquad \text{iff} \qquad \forall a,b.\ (a \ \ R \ \ b) \Longrightarrow \mathit{App}(f,a) \ \ G(a) \ \ \mathit{App}(g,b).$$

$$a \ \ \mathcal{P}wr(R) \ \ b \qquad \text{iff} \qquad \mathit{Rel}(a) = \mathit{Rel}(b) \in \mathrm{PER}(\mathcal{D}^\tau) \quad \text{and} \quad \mathit{Rel}(a) \subseteq R.$$

The good closure properties of PERs are well-known. The fact below is important in the soundness proof.

Fact 6.7. If $R \in \mathrm{PER}(\mathcal{D}^\tau)$ and $G(a) = G(b) \in \mathrm{PER}(\mathcal{D}^\upsilon)$ whenever $a \ \ R \ \ b$, then $\Pi(R,G) \in \mathrm{PER}(\mathcal{D}^{\tau\Rightarrow\upsilon})$.

Definition 6.8 (λ_{Power} environment model).
A λ_{Power} *environment model* for a structure $\mathcal{D}$ is an interpretation for $\mathcal{D}$ such that the following 9 conditions are satisfied.
For roughly-typable contexts $\Gamma, \Gamma_1, \Gamma_2$ and all $\eta \in \mathcal{D}^\Gamma$, $\eta_1 \in \mathcal{D}^{\Gamma_1}$, $\eta_2 \in \mathcal{D}^{\Gamma_2}$ with ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$,

CONST $[\![\kappa]\!]_\eta = \mathit{Const}(\kappa)$. VAR $[\![x]\!]_\eta = \eta^\Gamma(x)$.

CONST2 $R_{[\![\kappa]\!]_\eta} \in \mathrm{PER}(\mathcal{D}^\kappa)$. APP $[\![M\,N]\!]_\eta = \mathit{App}([\![M]\!]_\eta, [\![N]\!]_\eta)$.

FAMILY If for all a,b $(a \ \ R_{[\![A]\!]_\eta} \ \ b) \implies R_{[\![B]\!]_{\langle\eta,a\rangle}} = R_{[\![B]\!]_{\langle\eta,b\rangle}}$, then

$$R_{[\![\Pi x{:}A.B]\!]_\eta} = \Pi(R_{[\![A]\!]_\eta}, a \mapsto R_{[\![B]\!]_{\langle\eta,a\rangle}}).$$

SUBSET $R_{[\![\mathit{Power}(C)]\!]_\eta} = \mathcal{P}wr(R_{[\![C]\!]_\eta})$. In the above axioms we assume that for some τ, υ, $\Gamma \blacktriangleright x : \tau$, $\Gamma \blacktriangleright \Pi x{:}A.B : \mathsf{P}(\tau \Rightarrow \upsilon)$ and $\Gamma \blacktriangleright \mathit{Power}(C) : \mathsf{P}(\mathsf{P}(\tau))$.

ABS If $\Gamma \blacktriangleright A : \mathsf{P}(\tau)$, $\Gamma, x : A \blacktriangleright B : \mathsf{P}(\upsilon)$, and $\Gamma, x : A \blacktriangleright M, N : \tau$, then If $\forall d,e.$ $d \ \ R_{[\![A]\!]_\eta} \ \ e \implies [\![M]\!]_{\langle\eta,d\rangle} \ \ R_{[\![B]\!]_{\langle\eta,d\rangle}} \ \ [\![N]\!]_{\langle\eta,e\rangle}$, then

$$\forall d,e.\ \ d \ \ R_{[\![A]\!]_\eta} \ \ e \implies \mathit{App}([\![\lambda x{:}A.M]\!]_\eta, d) \ \ R_{[\![B]\!]_{\langle\eta,d\rangle}} \ \ [\![N]\!]_{\langle\eta,e\rangle}.$$

THIN If Φ is a renaming on $Dom(\Gamma_1)$ such that $\Phi(\Gamma_1) \subseteq \Gamma_2$, then

$$[\![\Phi(\Gamma_1) \blacktriangleright \Phi(M) : \tau]\!]_{\eta_1} = [\![\Gamma_2 \blacktriangleright M : \tau]\!]_{\eta_2}$$

where $\Gamma \blacktriangleright M : \tau$.

SUBST If $\Gamma_1 \equiv \Gamma, x : A, \Gamma'$, $\Gamma_2 \equiv \Gamma, \Gamma'[N/x]$ where $\Gamma \blacktriangleright N : \upsilon$ and $\Gamma \blacktriangleright A : \mathsf{P}(\upsilon)$; then

$$[\![\Gamma_1 \blacktriangleright M]\!]_{\eta_1} = [\![\Gamma_2 \blacktriangleright M[N/x]]\!]_{\eta_2}$$

provided $\Gamma_1 \blacktriangleright M : \tau$, ${\eta_1}^{\Gamma_1}(x) = [\![\Gamma \blacktriangleright N]\!]_{{\eta_1}|_x} \in \mathit{Rel}([\![\Gamma \blacktriangleright A]\!]_{{\eta_1}|_x})$. □

Axioms CONST, VAR, APP are standard in the model-definition for untyped and simply-typed λ-calculus [30]. The axiom CONST2 requires that atomic types denote PERs. The axioms FAMILY and SUBSET define the extension of the denotation of types of the form $\Pi x{:}A.B$ and $\mathit{Power}(C)$.

The axiom ABS ensures the soundness of the three equality rules which mention the λ-constructor.

Thinning THIN and substitution SUBST are fundamental properties that any model should satisfy; the latter need be satisfied only when the environment is updated with a value in the domain of the PER which interprets the type A added to the context. Proposition 5.3 ensures that THIN and SUBST are meaningful.

Soundness for models is proved in Section 7. First we revisit the example structures.

Full hierarchy model

Example 6.9 (Full hierarchy model). We define an interpretation by:

$$
\begin{array}{lcl}
\mathit{Rel}(A) & = & A \\
[\![\Gamma \blacktriangleright x : \tau \;\;]\!]_\eta = & \;\; & \eta^\Gamma(x) \\
[\![\Gamma \blacktriangleright \kappa : \mathsf{P}(\kappa)]\!]_\eta & = & R_\kappa \\
[\![\Gamma \blacktriangleright \lambda x{:}A.\, M : \tau \Rightarrow \upsilon]\!]_\eta & = & \text{the function } f : \mathcal{D}^\tau \to \mathcal{D}^\upsilon \text{ defined by} \\
 & & \quad f(n) = [\![\Gamma, x : A \blacktriangleright M : \upsilon]\!]_{\langle \eta, n \rangle} \\
 & & \text{for all } n \in \mathcal{D}^\tau. \\
[\![\Gamma \blacktriangleright M\,N : \upsilon]\!]_\eta & = & \mathit{App}([\![\Gamma \blacktriangleright M : \tau \Rightarrow \upsilon]\!]_\eta, [\![\Gamma \blacktriangleright N : \tau]\!]_\eta) \\
[\![\Gamma \blacktriangleright \Pi x{:}A.\, B : \mathsf{P}(\tau \Rightarrow \upsilon)]\!]_\eta & = & \Pi(R_{[\![A]\!]_\eta}, a \mapsto R_{[\![B]\!]_{\langle \eta, a \rangle}}) \\
[\![\Gamma \blacktriangleright \mathit{Power}(A) : \mathsf{P}(\mathsf{P}(\tau))]\!]_\eta & = & \mathcal{P}wr(R_{[\![A]\!]_\eta})
\end{array}
$$

□

Lemma 6.10. *The interpretation defined in Example 6.9 is a model of* λ_{Power}.

Proof We must check two things. First, that the interpretation function is type correct, in the sense that $[\![\Gamma \blacktriangleright M : \tau]\!]_\eta \in \mathcal{D}^\tau$ for all $\eta \in \mathcal{D}^\Gamma$; this is easy. Second, we must check that the model axioms are satisfied. All follow from the definitions, except SUBST and THIN which are proved by induction on rough-typing derivations. □

Here, $R_{[\![\Pi x:A.\,B]\!]_\eta}$ is not automatically a PER, since the uniformity condition that $a \;\; R_{[\![A]\!]_\eta} \;\; b \implies R_{[\![B]\!]_{\langle \eta, a \rangle}} = R_{[\![B]\!]_{\langle \eta, b \rangle}}$ may fail. For example, if $B \equiv zx$, the "rough-soundness" requirement that $\eta(z) \in \mathcal{D}^{\tau \Rightarrow \mathsf{P}(\upsilon)}$ does not force the value of z at one element of $\mathcal{D}^\tau$ to be related to the value at another. This is why we generalized to relations. PERs are only guaranteed for *well-formed* terms.

In the interpretation of Example 6.9, the meaning of the type A is *ignored* in the interpretation of $\lambda x{:}A.\, M$; we rely on $[\![M]\!]_{\langle \eta, e \rangle}$ being defined for every $e \in \mathcal{D}^\tau$. This is where the next example differs.

Full hierarchy partial function model

Example 6.11 (Full hierarchy partial function model). An interpretation in the partial structure can be defined as in Example 6.9 except for:

$$\mathsf{Rel}(A) \quad = \quad \begin{cases} A & \text{if } A \neq \diamond \\ \varnothing & \text{otherwise.} \end{cases}$$

$[\![\Gamma \blacktriangleright \lambda x{:}A.\, M : \tau \Rightarrow \upsilon]\!]_\eta$ = the partial function $f : \mathcal{D}^\tau \rightharpoonup \mathcal{D}^\upsilon$ defined by

$$f(n) = \begin{cases} [\![\Gamma, x : A \blacktriangleright M : \upsilon]\!]\langle \eta, n\rangle & \text{if } n \in [\![A]\!]_\eta \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \square$$

The partial function interpretation pays attention to the advertised domains of λ-abstractions, so that $[\![\lambda x{:}A.\, M]\!]_\eta$ is undefined for values $e \in \mathcal{D}^\tau$ which are not in $\mathsf{Rel}([\![A]\!]_\eta)$. Moreover, if a function variable is declared as having a type $\Pi x{:}A.\, B$, it need not be defined outside $[\![A]\!]$. This is why we add an error value $\diamond$ to each of the domains.

This construction is a more formal way of capturing the usual "partial definition" method of giving the interpretation function for dependent type-systems, the method first used by [37][10]. With a partial definition, the error element $\diamond$ is delegated to the meta-language, and one proves that the definition is meaningful for any well-typed term by induction on typing derivations. By contrast, the approach here asks for a *pre-interpretation* of roughly-typable terms, some of which are not well-typed in the full system. (Pitts [33] has another careful approach; he inductively defines a relation between a term and its denotation. The soundness proof shows that this relation amounts to a total function on well-typed terms.)

The partial function model of Example 6.11 doesn't behave quite as we might hope. It would be nice to establish the correctness property that, under some assumptions about the environment η, well-typed terms do not denote $\diamond$:

$$\Gamma \rhd M : A \qquad \Longrightarrow \qquad [\![M]\!]_\eta \neq \diamond.$$

We could prove this as a corollary of the soundness theorem (Theorem 7.5 below), if we had designed the interpretation so that for any type A, $\diamond \notin dom(R_{[\![A]\!]_\eta})$. However, this fails because of empty types. If B denotes an empty type, so $R_{[\![B]\!]_\eta} = \varnothing$, then $[\![\Pi x{:}B.\, C]\!]$ is the total relation which relates every element of $\mathcal{D}^{\tau \Rightarrow \upsilon}$ to every other.

This can't be fixed just by changing the definition of $[\![\Pi x{:}B.\, C]\!]_\eta$ in the interpretation; instead the model axiom FAMILY has to be modified to be "strict" — made aware of the error element $\diamond$. It would become:

$$f \;\; R_{[\![\Pi x:A.\,B]\!]_\eta} \;\; g$$
$$\text{iff}$$
$$\begin{cases} f \neq \diamond, g \neq \diamond \\ \forall a, b.\, (a \;\; R_{[\![A]\!]_\eta} \;\; b) \Longrightarrow \mathsf{App}(f, a) \;\; R_{[\![B]\!]_{\langle \eta, a\rangle}} \;\; \mathsf{App}(g, b) \end{cases}$$

And we would add the condition that $[\![\Pi x{:}A.\, B]\!]_\eta = \diamond$ if either $[\![A]\!]_\eta = \diamond$ or $[\![A]\!]_{\langle \eta, a\rangle} = \diamond$ for some $a \in R_{[\![A]\!]_\eta}$, to make the Π-constructor "strict". Then the soundness proof in Theorem 7.5 could be replayed to establish the correctness property above.

6.3.1 Term model

The term model is defined using the equational theory of the full system. This demonstrates the "externalisation" of equality in this model construction: usually one would use quotients of terms to construct a term model.

[10] and in fact independently by [34] for ASL+.

Example 6.12 (Term model). An interpretation in the term structure $\mathcal{T}_\Gamma$ can be defined by:

$$\begin{aligned} [\![\Gamma \blacktriangleright M : \tau]\!]_\eta &= \eta^\Gamma(M) \\ \mathsf{Rel}(A) &= \{ (M,N) \mid \Gamma \rhd M = N : A \} \end{aligned}$$

Where $\eta^\Gamma(M)$ denotes the result of substituting each free variable x in M for $\eta^\Gamma(x)$.

Unfortunately there is a stumbling point in trying to prove that the term model defined above in fact satisfies the model conditions. One way to fix this (suggested by recent research into pre-logical relations [24]) might be to relax the model axioms FAMILY and SUBSET to replace = by $\subseteq$, and add extra conditions that the model contains enough functions and subsets.

Conjecture 6.13. *The interpretation defined in Example 6.12 is a model of* λ_{Power}.

Proof Conjecture 4.12 is used for the right-to-left direction of SUBSET and Conjecture 4.11 is used for the right-to-left direction of FAMILY. □

7 Soundness

This section is taken up by a proof of soundness for any λ_{Power} interpretation which satisfies the model axioms. Specifically, when $\Gamma \rhd M : A$, then $[\![M]\!]_\eta$ is in the domain of the relation $R_{[\![A]\!]_\eta}$, and when $\Gamma \rhd M = N : A$, then $[\![M]\!]_\eta$ is related to $[\![N]\!]_\eta$ by $R_{[\![A]\!]}$. Moreover, $\mathsf{Rel}([\![A]\!]_\eta)$ is a PER on $\mathcal{D}^\tau$, where $\Gamma \blacktriangleright A : \mathsf{P}(\tau)$.

But we can only expect soundness if the environment η satisfies the context in a suitable way. The interpretation of a context Γ is defined by combining the interpretations of its components.

Let S and T be sets, $R \in \mathrm{REL}(S)$, and $G \in \mathit{dom}(R) \to \mathrm{REL}(T)$. Then we define $\Sigma(R,G) \in \mathrm{REL}(S \times T)$ by:

$$p \ \Sigma(R,G) \ q \qquad \text{iff} \qquad \pi_1(p) \ R \ \pi_1(q) \quad \text{and} \quad \pi_2(p) \ G(\pi_1(p)) \ \pi_2(q)$$

Fact 7.1. If $R \in \mathrm{PER}(S)$ and $G(a) = G(b) \in \mathrm{PER}(T)$ whenever $a \ R \ b$, then $\Sigma(R,G) \in \mathrm{PER}(S \times T)$.

Definition 7.2 (Interpretation of contexts).

- Given a model for $\mathcal{D}$ and a roughly-typable context Γ, we define $[\![\Gamma]\!] \in \mathrm{REL}(\mathcal{D}^\Gamma)$ by induction on Γ,

$$\begin{aligned} [\![\langle\rangle]\!] &= \{ (\star, \star) \} \\ [\![\Gamma', x : A]\!] &= \Sigma([\![\Gamma']\!], \eta \mapsto R_{[\![\Gamma' \blacktriangleright A : \mathsf{P}(\tau)]\!]_\eta}.) \end{aligned}$$

- We say $\eta_1, \eta_2 \in \mathcal{D}^\Gamma$ are *related environments satisfying* Γ iff $\eta_1 \ [\![\Gamma]\!] \ \eta_2$.

Fact 7.3. If Γ is roughly-typable and $\eta_1 \ [\![\Gamma]\!] \ \eta_2$, then for all $x \in \mathit{Dom}(\Gamma)$,

$$\eta_1^\Gamma(x) \ \ R_{[\![\Gamma|_x \blacktriangleright \Gamma(x) : \mathsf{P}(\tau)]\!]_{\eta_1|_x}} \ \ \eta_2^\Gamma(x). \qquad \square$$

Lemma 7.4. *Suppose that* $R_{[\![A]\!]_\eta} \in \mathrm{PER}(\mathcal{D}^\tau)$ *and that* $d \ R_{[\![A]\!]_\eta} \ e \implies R_{[\![B]\!]_{\langle \eta, d \rangle}} = R_{[\![B]\!]_{\langle \eta, e \rangle}} \in \mathrm{PER}(\mathcal{D}^\upsilon)$ *for all* d, e. *Then the following properties hold in any model:*

WEAK-EXT *If* $\forall d, e.\ d\ R_{[\![A]\!]_\eta}\ e \Longrightarrow [\![M]\!]_{\langle \eta, d\rangle}\ \ R_{[\![B]\!]_{\langle \eta, d\rangle}}\ \ [\![N]\!]_{\langle \eta, e\rangle}$, *then*

$$\forall d, e.\ \ d\ R_{[\![A]\!]_\eta}\ e \Longrightarrow \mathsf{App}([\![\lambda x{:}A.\, M]\!]_\eta, d)\ \ R_{[\![B]\!]_{\langle \eta, d\rangle}}\ \ \mathsf{App}([\![\lambda x{:}A.\, N]\!]_\eta, e)$$

ETA *If* $\forall d, e.\ d\ R_{[\![A]\!]_\eta}\ e \Longrightarrow \mathsf{App}([\![M]\!]_\eta, d)\ \ R_{[\![B]\!]_{\langle \eta, d\rangle}}\ \ \mathsf{App}([\![N]\!]_\eta, e)$, *then*

$$\forall d, e.\ \ d\ R_{[\![A]\!]_\eta}\ e \Longrightarrow \mathsf{App}([\![\lambda x{:}A.\, M\, x]\!]_\eta, d)\ \ R_{[\![B]\!]_{\langle \eta, d\rangle}}\ \ \mathsf{App}([\![N]\!]_\eta, e)$$

Proof WEAK-EXT follows from ABS using symmetry, transitivity and the assumption. ETA follows from ABS using the axioms THIN and APP. □

Theorem 7.5 (Soundness for models).

1. *If* $\triangleright\ \Gamma$ *then* $[\![\Gamma]\!] \in \mathrm{PER}(\mathcal{D}^\Gamma)$.
2. *If* $\Gamma\ \triangleright\ M : A$, *then* $\forall \eta_1, \eta_2 \in \mathcal{D}^\Gamma$,
$$\eta_1\ \ [\![\Gamma]\!]\ \ \eta_2 \Longrightarrow [\![M]\!]_{\eta_1}\ \ R_{[\![A]\!]_{\eta_1}}\ \ [\![M]\!]_{\eta_2}.$$
3. *If* $\Gamma\ \triangleright\ M = N : A$, *then* $\forall \eta_1, \eta_2 \in \mathcal{D}^\Gamma$,
$$\eta_1\ \ [\![\Gamma]\!]\ \ \eta_2 \Longrightarrow [\![M]\!]_{\eta_1}\ \ R_{[\![A]\!]_{\eta_1}}\ \ [\![N]\!]_{\eta_2}.$$

Proof By induction on derivation heights, using the model conditions. At a couple of points in the proof, we assume that any derivation of $\Gamma\ \triangleright\ M : A$ or $\Gamma\ \triangleright\ M = N : A$ has a shorter derivation of $\Gamma\ \triangleright\ A\ \mathsf{type}$. This is needed to use the induction hypothesis to show that

$$R_{[\![A]\!]_{\eta_1}} = R_{[\![A]\!]_{\eta_2}} \in \mathrm{PER}(\mathcal{D}^\tau)$$

where $\Gamma\ \blacktriangleright\ A : \mathsf{P}(\tau)$. The assumption is justified because any derivation has a corresponding derivation in a modified system with "strong" rules with subderivations $\Gamma\ \triangleright\ A\ \mathsf{type}$ in the premises. We must argue that the system with strong rules is equivalent. This can be done using an intermediate system with substitution as a rule, Proposition 4.7 and an argument similar to that in [38]. Alternatively, the equality above can be added to the statement of parts 2 and 3 of the theorem, at the cost of more work in the proof.

This proof ends on page 39. Instead of (SUB), we prove the soundness of the rule (EQ-SUB) (see 4.9).

1. **Case** (EMPTY): Then $[\![\langle\rangle]\!] = \{\,(\star, \star)\,\} \in \mathrm{PER}(\{\,\star\,\})$ by definition.

   Case (EXTEND): We have $\triangleright\ \Gamma', x : A$. By the induction hypothesis and the premise $\Gamma'\ \triangleright\ A\ \mathsf{type}$, we have
   $$R_{[\![A]\!]_{\eta_1}} \quad = \quad R_{[\![A]\!]_{\eta_2}} \quad \in \quad \mathrm{PER}(\mathcal{D}^\tau),$$
   for all η_1, η_2 such that $\eta_1\ \ R_{[\![\Gamma']\!]}\ \ \eta_2$, where $\Gamma'\ \blacktriangleright\ A : \mathsf{P}(\tau)$. By the induction hypothesis for the shorter derivation of $\triangleright\ \Gamma'$, $[\![\Gamma']\!] \in \mathrm{PER}(\mathcal{D}^\Gamma)$. So by Proposition 7.1, $[\![\Gamma', x : A]\!] \in \mathrm{PER}(\mathcal{D}^\Gamma)$ as required.

2. **Case** (ATOMIC): We have $\kappa : \mathsf{Power}(\kappa)$, so by CONST, we must show that
   $$\mathsf{Const}(\kappa)\ \ R_{[\![\mathsf{Power}(\kappa)]\!]_{\eta_1}}\ \ \mathsf{Const}(\kappa)$$
   which by SUBSET holds iff
   $$R_{\mathsf{Const}(\kappa)} = R_{\mathsf{Const}(\kappa)} \in \mathrm{PER}(\mathcal{D}^\kappa) \quad \text{and} \quad R_{\mathsf{Const}(\kappa)} \subseteq R_{\mathsf{Const}(\kappa)}$$
   which follows by CONST2 and tautology.

Case (VAR): We have $\Gamma \rhd x : \Gamma(x)$, so by VAR, we must show that

$$\eta_1(x) \;\; R_{[\![\Gamma \rhd \Gamma(x)]\!]_{\eta_1}} \;\; \eta_2(x)$$

which follows by the assumption that $\eta_1 \;\; [\![\Gamma]\!] \;\; \eta_2$ using Proposition 7.3 and THIN, since ${\eta_1|_x}^{\Gamma|_x} \subseteq {\eta_1}^{\Gamma}$.

Case (SUB): By the induction hypothesis,

$$[\![M]\!]_{\eta_1} \;\; R_{[\![A]\!]_{\eta_1}} \;\; [\![M]\!]_{\eta_2}$$

$$[\![A]\!]_{\eta_1} \;\; R_{[\![\mathit{Power}(B)]\!]_{\eta_1}} \;\; [\![A]\!]_{\eta_2}$$

by SUBSET,

$$R_{[\![A]\!]_{\eta_1}} \subseteq R_{[\![B]\!]_{\eta_1}}$$

hence

$$[\![M]\!]_{\eta_1} \;\; R_{[\![B]\!]_{\eta_1}} \;\; [\![M]\!]_{\eta_2}.$$

Case (REFL): We must show

$$[\![M]\!]_{\eta_1} \;\; R_{[\![\Pi \vec{x}:\vec{A}.\ \mathit{Power}(M\,\vec{x})]\!]_{\eta_1}} \;\; [\![M]\!]_{\eta_2}$$

Which, expanding and using FAMILY, holds iff for all $\vec{d}, \vec{e}$ such that $\langle \eta_1, \vec{d} \rangle \;\; [\![\Gamma, \vec{x} : \vec{A}]\!] \;\; \langle \eta_2, \vec{e} \rangle$,

$$\begin{array}{l} \mathit{App}([\![M]\!]_{\eta_1},\ \vec{d}) \\ \quad [\![\Gamma, \vec{x} : \vec{A} \blacktriangleright \mathit{Power}(M\,\vec{x})]\!]_{\langle \eta_1, \vec{d} \rangle} \\ \mathit{App}([\![M]\!]_{\eta_2},\ \vec{e}) \end{array} \tag{2}$$

which by SUBSET holds iff

$$R_{\mathit{App}([\![M]\!]_{\eta_1},\, \vec{d})} \quad = \quad R_{\mathit{App}([\![M]\!]_{\eta_2},\, \vec{e})} \quad \in \quad \mathrm{PER}(\mathcal{D}^{\tau}) \tag{2}$$

and

$$R_{\mathit{App}([\![M]\!]_{\eta_1},\, \vec{d})} \subseteq R_{[\![M\,\vec{x}]\!]_{\langle \eta_1, \vec{d} \rangle}} \tag{3}$$

(3) follows from APP, VAR and THIN. To show (2), we use the induction hypothesis, to give:

$$[\![M]\!]_{\eta_1} \;\; R_{[\![\Pi \vec{x}:\vec{A}.\ \mathit{Power}(B)]\!]_{\eta_1}} \;\; [\![M]\!]_{\eta_2}$$

hence by FAMILY,

$$\mathit{App}([\![M]\!]_{\eta_1},\ \vec{d}) \;\; [\![\Gamma, \vec{x} : \vec{A} \blacktriangleright \mathit{Power}(B)]\!]_{\langle \eta_1, \vec{d} \rangle} \;\; \mathit{App}([\![M]\!]_{\eta_2},\ \vec{e})$$

and then (2) follows by SUBSET, using the agreement of rough typing to show that both $\Gamma, \vec{x} : \vec{A} \blacktriangleright \mathit{Power}(M\,\vec{x}) : P(\tau)$ and $\Gamma, \vec{x} : \vec{A} \blacktriangleright \mathit{Power}(B) : P(\tau)$.

The condition for applying FAMILY in (1) can be shown to hold using the assumption about "strong" derivations, together with APP, VAR and SUBSET, and expanding the vector notation. Similar reasoning applies in other cases involving FAMILY.

Case (CONV): Follows easily by the induction hypothesis for the premises, using SUBSET.

Case (Π): We must show

$$[\![\Pi x{:}A.\,B]\!]_{\eta_1} \;\; R_{[\![\mathsf{Power}(\Pi x{:}A'.\,B')]\!]_{\eta_1}} \;\; [\![\Pi x{:}A.\,B]\!]_{\eta_2}$$

which by SUBSET holds iff

$$R_{[\![\Pi x:A.\,B]\!]_{\eta_1}} \quad = \quad R_{[\![\Pi x:A.\,B]\!]_{\eta_2}} \quad \in \quad \mathrm{PER}(\mathcal{D}^{\tau\to\upsilon}) \tag{1}$$

and

$$R_{[\![\Pi x:A.\,B]\!]_{\eta_1}} \subseteq R_{[\![\Pi x:A'.\,B']\!]_{\eta_1}} \tag{2}$$

By FAMILY,

$$R_{[\![\Gamma \rhd \Pi x:A.\,B]\!]_{\eta_1}} = \Pi(R_{[\![A]\!]_{\eta_1}}, a \mapsto R_{[\![B]\!]_{\langle \eta_1, a\rangle}}) \tag{3}$$

By the induction hypothesis for the shorter derivation of $\Gamma \rhd A$ *type* and SUBSET,

$$R_{[\![A]\!]_{\eta_1}} \quad = \quad R_{[\![A]\!]_{\eta_2}} \quad \in \quad \mathrm{PER}(\mathcal{D}^{\tau}) \tag{3}$$

and by the induction hypothesis for the premise $\Gamma, x : A \rhd B$ *type* and the SUBSET axiom again,

$$R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_1, a\rangle}} \quad = \quad R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_2, a\rangle}} \quad \in \quad \mathrm{PER}(\mathcal{D}^{\upsilon}) \tag{4}$$

for all $a \in R_{[\![A]\!]_{\eta_1}}$, so by (4), (5), FAMILY and Proposition 6.7 the equation (1) holds. $R_{[\![\Gamma \rhd \Pi x:A.\,B]\!]_{\eta_1}} = R_{[\![\Gamma \rhd \Pi x:A.\,B]\!]_{\eta_2}}$. [11]

To show the inclusion in (2), let $f, g \in \mathcal{D}^{\tau\to\upsilon}$ and suppose

$$f \;\; R_{[\![\Pi x:A.\,B]\!]_{\eta_1}} \;\; g. \tag{6}$$

We must show

$$f \;\; R_{[\![\Pi x:A'.\,B']\!]_{\eta_1}} \;\; g \tag{7}$$

which by FAMILY is implied by

$$\begin{array}{l} \forall d, e.\, d \;\; R_{[\![A']\!]_{\eta_1}} \;\; e \\ \quad \Longrightarrow \quad \mathsf{App}(f, d) \;\; R_{[\![\Gamma, x:A' \rhd B']\!]_{\langle \eta_1, d\rangle}} \;\; \mathsf{App}(g, e) \end{array} \tag{8}$$

Suppose that $d \;\; R_{[\![A']\!]_{\eta_1}} \;\; e$ for some d, e. By induction hypothesis for the premise $\Gamma \rhd A' : \mathsf{Power}(A)$ and SUBSET,

$$R_{[\![A']\!]_{\eta_1}} \subseteq R_{[\![A]\!]_{\eta_1}}$$

so

$$\mathsf{App}(f, d) \;\; R_{[\![\Gamma, x:A \rhd B]\!]_{\eta_1}} \;\; \mathsf{App}(g, e)$$

using FAMILY and (6). By induction hypothesis for the premise $\Gamma, x : A' \rhd B : \mathsf{Power}(B')$ and SUBSET,

$$R_{[\![\Gamma, x:A' \rhd B]\!]_{\langle \eta_1, d\rangle}} \subseteq R_{[\![\Gamma, x:A' \rhd B']\!]_{\langle \eta_1, d\rangle}}$$

hence

$$\mathsf{App}(f, d) \;\; R_{[\![\Gamma, x:A' \rhd B']\!]_{\eta_1}} \;\; \mathsf{App}(g, e)$$

as required.

[11] Since value of $\Pi(R_{[\![A]\!]_\eta}, a \mapsto R_{[\![B]\!]_{\langle \eta, a\rangle}})$ only depends on $[\![\Gamma, x : A \rhd B]\!]_{\langle \eta, a\rangle}$ for values $a \in R_{[\![A]\!]_{\eta_1}}$

Case (*Power*): We must show

$$[\![\mathsf{Power}(A)]\!]_{\eta_1} \;\; R_{[\![\mathsf{Power}^2(B)]\!]_{\eta_1}} \;\; [\![\mathsf{Power}(A)]\!]_{\eta_2}$$

which by SUBSET holds iff

$$R_{[\![\mathsf{Power}(A)]\!]_{\eta_1}} \quad = \quad R_{[\![\mathsf{Power}(A)]\!]_{\eta_2}} \quad \in \quad \mathrm{PER}(\mathcal{D}^{P(\tau)}) \tag{1}$$

$$R_{[\![\mathsf{Power}(A)]\!]_{\eta_1}} \subseteq R_{[\![\mathsf{Power}(B)]\!]_{\eta_1}} \tag{2}$$

By the induction hypothesis and SUBSET, we have

$$R_{[\![A]\!]_{\eta_1}} \quad = \quad R_{[\![A]\!]_{\eta_2}} \in \mathrm{PER}(\mathcal{D}^{\tau}) \tag{3}$$

$$R_{[\![A]\!]_{\eta_1}} \subseteq R_{[\![B]\!]_{\eta_2}} \tag{4}$$

so (1) follows from (3) using SUBSET. To show the inclusion (2), suppose that $s \;\; R_{[\![\mathsf{Power}(A)]\!]_{\eta_1}} \;\; t$. Then by SUBSET and (3), (4),

$$R_s = R_t \subseteq R_{[\![B]\!]_{\eta_2}}$$

which implies (2) by SUBSET.

3. **Case** (EQ-REFL): By the induction hypothesis.

   Case (EQ-SYM): We must show

   $$[\![N]\!]_{\eta_1} \;\; R_{[\![A]\!]_{\eta_1}} \;\; [\![M]\!]_{\eta_2}$$

   We need to appeal to the premise $\Gamma \rhd A \;\; \mathit{type}$ for the "strong" rule which by induction hypothesis guarantees that $R_{[\![A]\!]_{\eta_1}} = R_{[\![A]\!]_{\eta_2}} \in \mathrm{PER}(\mathcal{D}^{\tau})$ for some τ. Now by induction hypothesis for the premise of the rule,

   $$[\![M]\!]_{\eta_2} \;\; R_{[\![A]\!]_{\eta_2}} \;\; [\![N]\!]_{\eta_1}$$

   hence the result by symmetry of $R_{[\![A]\!]_{\eta_2}}$.

   Case (EQ-TRANS): Similar to the proof for (EQ-SYM), using SUBSET to show that $R_{[\![A]\!]_{\eta_1}}$ is transitive.

   Case (EQ-λ): We must show

   $$[\![\lambda x{:}A.\, M]\!]_{\eta_1} \;\; R_{[\![\Pi x:A.\, B]\!]_{\eta_1}} \;\; [\![\lambda x{:}A.\, M']\!]_{\eta_2}$$

   The induction hypothesis gives

   $$\begin{array}{l} \forall d, e.\, d \;\; [\![A]\!]_{\eta_1} \;\; e \\ \quad \Longrightarrow \quad [\![\Gamma, x : A \rhd M]\!]_{\langle \eta_1, d \rangle} \;\; R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_1, d \rangle}} \;\; [\![\Gamma, x : A \rhd M']\!]_{\langle \eta_2, e \rangle} \end{array}$$

   By WEAK-EXT, we get

   $$\begin{array}{l} \forall d, e.\, d \;\; [\![A]\!]_{\eta_1} \;\; e \\ \quad \Longrightarrow \quad \mathsf{App}([\![\lambda x{:}A.\, M]\!]_{\eta_1}, d) \;\; R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_1, d \rangle}} \;\; \mathsf{App}([\![\lambda x{:}A.\, M']\!]_{\eta_2}, e) \end{array}$$

   from which the result follows by FAMILY.

Case (EQ-APP): By the APP axiom, we must show

$$App(\llbracket M \rrbracket_{\eta_1}, \llbracket N \rrbracket_{\eta_1}) \;\; R_{\llbracket B[N/x] \rrbracket_{\eta_1}} \;\; App(\llbracket M' \rrbracket_{\eta_2}, \llbracket N' \rrbracket_{\eta_2})$$

Using the induction hypothesis for the premises, we have

$$\llbracket M \rrbracket_{\eta_1} \;\; R_{\llbracket \Pi x{:}A.\, B \rrbracket_{\eta_1}} \;\; \llbracket M' \rrbracket_{\eta_2}$$

$$\llbracket N \rrbracket_{\eta_1} \;\; R_{\llbracket A \rrbracket_{\eta_1}} \;\; \llbracket N' \rrbracket_{\eta_2}$$

and so by the axiom FAMILY, we have

$$App(\llbracket M \rrbracket_{\eta_1}, \llbracket N \rrbracket_{\eta_1}) \;\; R_{\llbracket B \rrbracket_{\langle \eta_1, \llbracket N \rrbracket_{\eta_1} \rangle}} \;\; App(\llbracket M' \rrbracket_{\eta_2}, \llbracket N' \rrbracket_{\eta_2})$$

But using SUBST,

$$\llbracket B \rrbracket \langle \eta_1, \llbracket N \rrbracket_{\eta_1} \rangle \quad = \quad \llbracket B[N/x] \rrbracket_{\eta_1}$$

so we are done.

Case (EQ-β): We must show

$$\llbracket (\lambda x{:}A.\, M)\, N \rrbracket_{\eta_1} \;\; R_{\llbracket B[N/x] \rrbracket_{\eta_1}} \;\; \llbracket M'[N'/x] \rrbracket_{\eta_2} \tag{1}$$

By the induction hypothesis,

$$\llbracket \Gamma,\, x : A \rhd M \rrbracket \langle \eta_1, d \rangle \;\; R_{\llbracket \Gamma, x:A \rhd B \rrbracket \langle \eta_1, d \rangle} \;\; \llbracket \Gamma,\, x : A \rhd M' \rrbracket \langle \eta_2, e \rangle$$

for all d, e such that $d \;\; R_{\llbracket \Gamma \rhd A \rrbracket_{\eta_1}} \;\; e$, and

$$\llbracket N \rrbracket_{\eta_1} \;\; R_{\llbracket A \rrbracket_{\eta_1}} \;\; \llbracket N' \rrbracket_{\eta_2}$$

using SUBST,

$$R_{\llbracket \Gamma \rhd B[N/x] \rrbracket_{\eta_1}} = R_{\llbracket \Gamma, x:A \rhd B \rrbracket \langle \eta_1, \llbracket \Gamma \rhd N \rrbracket_{\eta_1} \rangle} \tag{2}$$

and

$$\llbracket \Gamma,\, x : A \rhd M' \rrbracket \langle \eta_2, \llbracket N' \rrbracket_{\eta_2} \rangle = \llbracket \Gamma \rhd M'[N'/x] \rrbracket_{\eta_2} \tag{3}$$

By the ABS axiom,

$$App(\llbracket \lambda x{:}A.\, M \rrbracket_{\eta_1}, \llbracket N \rrbracket_{\eta_1}) \;\; R_{\llbracket \Gamma, x:A \rhd B \rrbracket \langle \eta_1, \llbracket N \rrbracket_{\eta_1} \rangle} \;\; \llbracket M' \rrbracket \langle \eta_2, \llbracket N' \rrbracket_{\eta_2} \rangle$$

which gives the result (1) using (2) and (3).

Case (EQ-η): We must show

$$\llbracket \lambda x{:}A.\, M\, x \rrbracket_{\eta_1} \;\; R_{\llbracket \Pi x:A.\, B \rrbracket_{\eta_1}} \;\; \llbracket N \rrbracket_{\eta_2}$$

By FAMILY, it suffices to show that

$$\begin{array}{l} \forall d, e.\, d \;\; R_{\llbracket \Gamma \rhd A \rrbracket_{\eta_1}} \;\; e \\ \quad \Longrightarrow \quad App(\llbracket \lambda x{:}A.\, M\, x \rrbracket_{\eta_1}, d) \;\; R_{\llbracket \Gamma, x:A \rhd B \rrbracket \langle \eta_1, d \rangle} \;\; App(\llbracket N \rrbracket_{\eta_2}, e) \end{array}$$

which follows by ETA and the induction hypothesis for the premise, using FAMILY. □

$$\forall d, e.\, d\ R_{[\![\Gamma \rhd A]\!]_{\eta_1}}\ e \\ \implies \mathsf{App}([\![M\,x]\!]_{\eta_1}, d)\ R_{[\![\Gamma, x{:}A \rhd B]\!]_{\langle \eta_1, d\rangle}}\ \mathsf{App}([\![N]\!]_{\eta_2}, e) \tag{3}$$

by the induction hypothesis,

$$[\![M]\!]_{\eta_1}\ R_{[\![\Pi x{:}A.\,B]\!]_{\eta_1}}\ [\![N]\!]_{\eta_2} \tag{4}$$

Case (EQ-SUB): We must show

$$[\![M]\!]_{\eta_1}\ R_{[\![B]\!]_{\eta_1}}\ [\![N]\!]_{\eta_2} \tag{1}$$

by the induction hypothesis,

$$[\![M]\!]_{\eta_1}\ R_{[\![A]\!]_{\eta_1}}\ [\![N]\!]_{\eta_2} \tag{2}$$

$$[\![A]\!]_{\eta_1}\ R_{[\![\mathsf{Power}(B)]\!]_{\eta_1}}\ [\![A']\!]_{\eta_2} \tag{3}$$

by SUBSET and (3),

$$R_{[\![A]\!]_{\eta_1}} \subseteq R_{[\![B]\!]_{\eta_1}}$$

hence (1) follows from (2).

Theorem 7.5 can be extended to prove that the admissible rules shown in Section 4 are sound in any model. As a special case of the theorem, the typing judgement is sound.

Corollary 7.6 (Soundness of Typing).
If $\Gamma \rhd M : A$, *then for all* η, $\eta \in dom[\![\Gamma]\!] \implies [\![M]\!]_\eta \in R_{[\![A]\!]_\eta}$.

8 Strong Normalization

In this section I show that $\beta\eta$-reduction is strongly normalizing on roughly-typed terms. By Proposition 5.6, all typable terms of λ_{Power} are roughly-typable, so we get strong normalization for λ_{Power} terms as a corollary.

The proof follows the well-known computability method due to Tait and Martin-Löf with amendments for power types. We use an extended language of terms,

$$\boldsymbol{T} \ ::= \ \cdots \ | \ \lfloor \boldsymbol{T} \rfloor$$

The new constructor is motivated by a reduction relation on terms with power type which is useful in type-checking algorithms, but we will not need the reduction here.[12] The properties of the rough typing system Section 5 are preserved when we add the rough typing rule:

$$\frac{\Gamma \blacktriangleright A : \mathsf{P}(\tau)}{\Gamma \blacktriangleright \lfloor A \rfloor : \tau}$$

Now some basic definitions.

[12] For the cognoscenti, the reduction relation includes the replacement $\lfloor \mathsf{Power}(A) \rfloor \longrightarrow_p A$. It is used when a type variable is promoted to its bound declared in the context, when we have to "strip" *Power*-constructors. This operation crops up when the well-known algorithms for subtyping [15, 13, 36] are re-expressed for power types.

Definition 8.1.

- A *canonical term* has one of the forms: κ, $\lambda x{:}A.M$, $\Pi x{:}A.B$, or $\mathsf{Power}(A)$.
- A *base term* is either a variable, or has the form $M\,N$ or $\lfloor M \rfloor$, where M is a base term.
- Weak-head reduction is the one-step reduction relation defined by:

$$\frac{M \longrightarrow_{\beta\eta} N}{M \longrightarrow_{wh} N} \qquad \frac{M \longrightarrow_{wh} M'}{M\,N \longrightarrow_{wh} M'\,N} \qquad \frac{M \longrightarrow_{wh} M'}{\lfloor M \rfloor \longrightarrow_{wh} \lfloor M' \rfloor}$$

  where $M \longrightarrow_{\beta\eta} N$ is an outermost single-step of β or η reduction.
- Finally, SN stands for the set of $\beta\eta$-strongly normalizing terms in $\boldsymbol{T}$.

A roughly-typable term is either canonical, a base term, or a term which can be weak-head reduced. This case distinction lies at the heart of the computability proof of strong normalization.

Definition 8.2. A set of pre-terms S is a *saturated set* iff

1. $S \subseteq \mathrm{SN}$.
2. If $M \in \mathrm{SN}$ is a base term, then $M \in S$.
3. If $M \in \mathrm{SN}$ is a weak-head redex and $M' \in S$ where $M \longrightarrow_{wh} M'$, then $M \in S$ too.

Next we define an interpretation of types by giving a term applicative structure of strongly normalizing pre-terms, and we define an interpretation of terms by substitution. I follow Section 6.1 for notation, although the language is extended here.

Definition 8.3.

- The term applicative structure S is defined by:

$$\begin{aligned} S^{\kappa} &= \mathrm{SN} \\ S^{\tau \Rightarrow \upsilon} &= \{\, M \in \boldsymbol{T} \;\mid\; \forall N \in S^{\tau}.\, M\,N \in S^{\upsilon} \,\} \\ S^{\mathsf{P}(\tau)} &= \{\, M \in \boldsymbol{T} \;\mid\; \lfloor M \rfloor \in S^{\tau} \,\} \end{aligned}$$

- When $\Gamma \blacktriangleright M : \tau$ and $\eta \in S^{\Gamma}$, then $[\![\Gamma \blacktriangleright M : \tau]\!]_{\eta} = \eta^{\Gamma}(M)$.

I haven't defined App and Const because the λ_{Power} model axioms aren't needed. Proposition 8.5 below says that $[\![\Gamma \blacktriangleright M : \tau]\!]_{\eta}$ satisfies the "rough soundness" requirement given as the first part of Definition 6.6. This is enough for the strong normalization result.

Lemma 8.4. *For all τ, the set S^{τ} is saturated.*

Proof By induction on the structure of τ. We consider the new case for power types, where $\tau \equiv \mathsf{P}(\upsilon)$ and S^{υ} is saturated by induction hypothesis. We must check the conditions to show that $S^{\mathsf{P}(\upsilon)}$ is saturated.

1. (1) Suppose $M \in S^{\mathsf{P}(\upsilon)}$. Then $\lfloor M \rfloor \in S^{\upsilon}$ and is SN by the induction hypothesis. Hence M is SN.
2. (2) Let M be a SN base term. Then so too is $\lfloor M \rfloor$, therefore $M \in S^{\mathsf{P}(\upsilon)}$.
3. (3) Suppose that $M \longrightarrow_{wh} M'$ and $M' \in S^{\mathsf{P}(\upsilon)}$. Then $\lfloor M' \rfloor \in S^{\upsilon}$ and by the induction hypothesis, $\lfloor M \rfloor \in S^{\upsilon}$ too, because $\lfloor M \rfloor \longrightarrow_{wh} \lfloor M' \rfloor$. Hence $M \in S^{\mathsf{P}(\upsilon)}$. □

Lemma 8.5. *Let $\Gamma \blacktriangleright M : \tau$. For all $\eta \in S^{\Gamma}$, $[\![\Gamma \blacktriangleright M : \tau]\!]_{\eta} \in S^{\tau}$.*

Proof By induction on the derivation of $\Gamma \blacktriangleright M : \tau$. □

Theorem 8.6 (Strong normalization for roughly-typed terms).
If $\Gamma \blacktriangleright M : \tau$ then M is $\beta\eta$-strongly normalizing.

Proof Immediate by Lemmas 8.4 and 8.5. □

Corollary 8.7 (Strong normalization for λ_{Power}).
If $\Gamma \rhd M : A$, then M is $\beta\eta$-strongly normalizing.

Proof Immediate by Proposition 5.6. □

9 Conclusions

This paper introduces the type system λ_{Power}, a predicative fragment of Cardelli's original power type system [9].

Power types provide a cunning way of dealing with the subtyping judgement at the same time as the typing judgement. At first sight it appears to be a simplification, because two separate concerns are combined into one. However, the generalisation which occurs from using $\mathsf{Power}(A)$ as both a term and a type leads to complication of the meta-theory.

The semantics of λ_{Power} is set-based, but uses partial equivalence relations to interpret equality. The subtyping relation induced by power types is understood as inclusion between PERs. In contrast to other semantics for subtyping or dependent types, the intended model is made by "carving out" from a classical set-hierarchy, without using a universal domain. Every term in λ_{Power} has a *rough type* which is either an atomic type, or one of the forms $\tau \Rightarrow \upsilon$ or $\mathsf{P}(\tau)$, where τ and υ are rough types. These rough types are used to structure the set hierarchy. The model definition is a collection of axioms which an interpretation must satisfy to be sound.

The important results established include the properties in Section 4, particularly type correctness, Proposition 4.8; the agreement with the rough typing system, Proposition 5.6; the soundness property for the semantics, Theorem 7.5; and the strong normalization result, given in Proposition 8.7. Unfortunately, there are still gaps in the meta-theory of λ_{Power}: ideally, we would like to prove a generation principle and thus prove subject reduction for λ_{Power}, which seems less straightforward than might be hoped (but no counterexamples have been found).

Acknowledgements. I'm especially grateful to Don Sannella who suggested that I study ASL+ [34], from which λ_{Power} grew. The work here first appeared in my thesis supervised by Sannella [2]. I am grateful to many people I discussed this work with during its evolution, including L. Cardelli, A. Compagnoni, H. Goguen, M. Hofmann, B. Pierce, and A. Tarklecki. More recently, the referees for CSL 2000 provided very useful and detailed comments.

References

[1] David Aspinall. Types, subtypes, and ASL+. In *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 906. Springer-Verlag, 1995. URL `ftp://ftp.dcs.ed.ac.uk/pub/da/TSASL+.ps.gz`.

[2] David Aspinall. *Type Systems for Modular Programs and Specification*. PhD thesis, Department of Computer Science, University of Edinburgh, 1997. URL `http://www.dcs.ed.ac.uk/home/da/thesis`.

[3] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In E. Clarke, editor, *Proceedings, Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 86–97, New Brunswick, New Jersey, 1996. IEEE Computer Society Press. URL `http://www.dcs.ed.ac.uk/home/da/papers/psub`.

[4] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[5] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[6] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[7] Kim Bruce and John C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings, Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–327, Albuquerque, New Mexico, January 19–22, 1992. ACM Press. URL `http://hypatia.dcs.qmw.ac.uk/authors/B/BruceKB/BrMitPOPL.dvi.Z`.

[8] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87:196–240, 1990.

[9] Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, California, January 13–15, 1988. ACM SIGACT-SIGPLAN, ACM Press.

[10] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[11] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[12] A. Compagnoni and H. Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS–LFCS–97–361, LFCS, Department of Computer Science, University of Edinburgh, July 1997.

[13] Adriana Beatriz Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Nijmegen Catholic University, 1995.

[14] Karl Crary. *Type-theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[15] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[16] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. URL `http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/92/ECS-LFCS-92-227/index.html`.

[17] Herman Geuvers. *Logics and Type Systems*. PhD thesis, Nijmegen Catholic University, 1993.

[18] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.

[19] Herman Geuvers and Benjarnin Werner. On the Church-Rosser property for expressive type systems and its consequences for their metatheoretic study. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 320–329, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[20] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, 1994.

[21] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[22] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1): 143–184, 1993. URL `http://www.cs.cmu.edu/ fp/elf-papers/jacm93.dvi.gz`.

[23] J. Hindley and J. Seldin, editors. *Introduction to Lambda-Calculus and Combinators*. London, 1987.

[24] Furio Honsell and Donald Sannella. Pre-logical relations. In *Proc. Computer Science Logic, CSL'99*, volume 1683 of *Lecture Notes in Computer Science*, pages 546–561. Springer, 1999. URL `ftp://ftp.dcs.ed.ac.uk/pub/dts/prelogrel.ps`.

[25] Bart Jacobs and Tom Melham. Translating dependent type theory into higher order logic. In M. Bezem and J.F.Groote, editors, *Typed Lambda Calculi and Applications, International Conference, March 1993, Utrecht, The Netherlands*, volume 664, pages 209–229. Springer-Verlag, 1993.

[26] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. A logic of subtyping (extended abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 292–299, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[27] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[28] Zhaohui Luo. Coercive subtyping in type theory. In *Proc. Computer Science Logic, CSL'96, Utrecht*, 1996. URL `http://www.dur.ac.uk/ dcs0zl/SUBTYPING96.dvi`.

[29] Ian A Mason. Hoare's Logic in the LF. Technical Report ECS-LFCS-87-32, LFCS, Department of Computer Science, University of Edinburgh, June 1987.

[30] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Formal Models and Semantics – Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365–458. Elsevier, 1990.

[31] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[32] Frank Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 315–328, May 1993.

[33] A. M. Pitts. Categorical logic. In *Handbook of Logic in Computer Science*, volume 6. OUP, 1996.

[34] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992. URL `http://www.dcs.ed.ac.uk/home/dts/pub/param.ps`.

[35] Donald Sannella and Andrzej Tarlecki. A kernel specification formalism with higher-order parameterisation. In H. Ehrig et al., editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 534, pages 274–296. Springer-Verlag, 1991.

[36] Martin Steffen and Benjamin Pierce. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1994. URL `http://www.cl.cam.ac.uk/users/bcp1000/ftp/fomega.ps.gz`.

[37] Thomas Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991.

[38] A. S. Troelstra. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science*, 51:1–26, 1987.