Subtyping with Power Types[⋆]

David Aspinall
http://www.dcs.ed.ac.uk/home/da

LFCS, University of Edinburgh, U.K.

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.
Keywords: **type theory, subtyping, dependent types.**

1 Introducing Power Types

Power types were introduced in a seminal paper by Cardelli [4]. 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 $A \leq B \ =_{\mathrm{def}}\ A : \mathsf{Power}(B)$. The rules for power types are chosen to make this definition sensible. Cardelli called the three basic rules power introduction, elimination and subtyping:

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

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.

⋆ Summary version. The full version [2] is available from my web page, address above.

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, defining $\Lambda\alpha \leq A.M \;=_{\mathrm{def}}\; \lambda\alpha{:}\,\mathsf{Power}(A).\,M$ and $\forall\alpha \leq A.B \;=_{\mathrm{def}}\; \Pi\alpha{:}\,\mathsf{Power}(A).\,B$. This is a simplification, since there is no need to add new constructs. (The work described here grew from a slightly different application: in ASL+ [1], subtyping models specification refinement, and $\Lambda X \leq SP.M$ is a parameterised specification which can be applied to any refinement of SP.) Unfortunately, Cardelli's full power type system is tricky to handle: it has impredicative polymorphism via the $\mathsf{Type} : \mathsf{Type}$ axiom along with other features, rendering it undecidable, inconsistent when viewed as a logic, and difficult to give a semantics to. Later work on Quest [6] used power *kinds* instead, where $\mathsf{Power}(A)$ does not enjoy the status of a type itself.

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 (Section 6). 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 7 summarizes.

2 Examples in $\boldsymbol{\lambda}_{\mathsf{Power}}$

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 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.

2.1 A simple programming example

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

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

Imagine that $\mathsf{Upto}\,n$ stands for the set $\{\, m \in \mathsf{nat} \;\;|\;\; m \leq n \,\}$, and $\mathsf{Perm}\,n$ is the set of permutations of $\{\, 1, \ldots, n \,\}$, which is a subset of the set of functions from

[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.

$\mathsf{Upto}\,n$ to $\mathsf{Upto}\,n$. The function $\mathsf{Invperm}\,n\,p$ 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:

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

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

$$\mathit{\Gamma}_{\mathrm{PERM}} \triangleright \mathsf{ApplyPerm} : \mathit{\Pi} n{:}\mathsf{nat}.\,(\mathsf{Perm}\,n) \to (\mathsf{Upto}\,n) \to (\mathsf{Upto}\,n).$$

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

2.2 Subtyping type operators and families

Systems of *higher-order subtyping* extend subtyping to type-constructors. The prototypical one is $F^{\omega}_{\leq}$ [5], in which one can declare a type variable ranging over type operators, $F \leq (\lambda\beta \leq \mathsf{nat}.\,\mathsf{List}(\beta \times \beta))$. A system with dependent types instead of polymorphism is $\lambda\mathrm{P}_{\leq}$ [3], in which one can declare a variable ranging over type families, $G \leq (\lambda x{:}\mathsf{nat}.\,\mathsf{Vec}_{\mathsf{nat}}(5 * x))$. In the first case, F ranges over constructors that map a subtype β of nat to a subtype of $\mathsf{List}(\beta \times \beta)$; 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:

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

$$\frac{\mathit{\Gamma} \triangleright H \leq J \qquad \mathit{\Gamma} \triangleright J\,C : K}{\mathit{\Gamma} \triangleright H\,C \leq J\,C} \qquad (\textsc{sub-app})$$

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

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

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

In λ_{Power}, there is no rule directly corresponding to ($\textsc{sub-}\lambda$). Indeed it is impossible to prove anything with the form $\mathit{\Gamma} \triangleright \lambda\alpha{:}K.\,A : \mathsf{Power}(C)$. 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.

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 $\mathit{\Pi}\alpha{:}K.\ \mathsf{Power}(H\,\alpha)$, since each application $F\,M$ must be a subtype of $H\,M$. This "η-like" expansion for Π-types works uniformly[2] and we can declare:

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

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

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

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.[3]

2.3 λ_{Power} as a logical framework

λ_{Power} is related to λP, which underlies the Edinburgh LF [9]. 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 $U =_{\mathrm{def}} \mathit{Power}(\upsilon)$. 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 [11, 3], but λ_{Power} goes beyond both these systems by allowing refinements of the universe U itself.

Gardner proposed doing this [8] to help adequacy proofs. She defined a framework ELF+ which distinguishes between terms that represent: object-level syntax, proof terms, and other terms. To emulate ELF+ in λ_{Power}, declare three subtypes: `Sort` : $\mathit{Power}(U)$, `Judge` : $\mathit{Power}(U)$, and `Type` : $\mathit{Power}(U)$.

An encoding where power types are useful is higher-order logic (HOL). Simple types τ of the form ι, o, and $\tau \Rightarrow \tau$ are encoded in an LF type `dom` : `Type`, with `i`, `o` : `dom`, $\Rightarrow$: `dom` $\to$ `dom` $\to$ `dom` and `obj` : `dom` $\to$ `Type`. HOL terms with domain τ are represented as elements of `obj`(τ). ELF+ improves this, showing `dom` and `obj` to be artifacts of the encoding, inhabiting `Type`, and typing `obj` : `dom` $\to$ `Sort`, showing that elements of `obj`(τ) correspond to object logic syntax. But in both LF and ELF+, the proliferation of `obj` quickly pollutes large terms. In λ_{Power}, we can remove it altogether and declare `dom` : Power(`Sort`). The

[2] This idea also appears in Crary's λK system which has power kinds [7].
[3] But *practical* effects on type-checking algorithms have not been investigated yet.
[4] Perhaps, moreover, λ_{Power} is conservative over λP under this translation.

mapping `obj` is now implicit; the representation of the logic becomes more concise, yet no less accurate. For example, the application term former becomes:

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

instead of

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

Although simple, it is important to emphasise that this example goes beyond many other subtyping proposals. Power types apply uniformly; other systems would have to be extended with sub-kinding to cope with this example.

3 The System $\boldsymbol{\lambda}_{\mathsf{Power}}$

Let $\mathbf{V}$ be a fixed countable infinite set of variables and $\mathcal{K}$ be a set of atomic type constants. The set $\mathbf{T}_{\mathcal{K}}$ of *pre-terms* is given by:

$$\mathbf{T} ::= \mathcal{K} \quad | \quad \mathbf{V} \quad | \quad \lambda\mathbf{V}{:}\mathbf{T}.\,\mathbf{T} \quad | \quad \mathbf{T}\,\mathbf{T} \quad | \quad \Pi\mathbf{V}{:}\mathbf{T}.\,\mathbf{T} \quad | \quad \mathsf{Power}(\mathbf{T})$$

(writing $\mathbf{T}$ as short for $\mathbf{T}_{\mathcal{K}}$). For meta-variables I use $x, y, \ldots \in \mathbf{V}$, $\kappa, \ldots \in \mathcal{K}$, and $A, B, \ldots, M, N, \ldots \in \mathbf{T}$. Usual conventions are used for writing pre-terms. 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. Terms and types are defined via three judgement forms:

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

These judgements are defined simultaneously by the rules shown at the end of the paper. The system λ_{Power} is close to a predicative fragment of Cardeilli's original system [4]; 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 judgement $\Gamma \triangleright A : \mathsf{Power}(B)$ 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). Most rules are standard. 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:

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

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.

Equality rules (Figure 2). These rules are standard.

Definition 3.1 (Terms, types and subtypes). *We say that* M *is a* Γ-term *if for some* A, $\Gamma \rhd M : A$, A *is a* Γ-type *if for some* B, $\Gamma \rhd A : \mathsf{Power}(B)$, *and* 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. There are three derived judgement forms:

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

Section 4 shows that these definitions make sense.

4 Properties of $\boldsymbol{\lambda}_{\mathsf{Power}}$

The development begins with showing derivability of several rules: that the induced subtype relation is a pre-order, and that type equality is reflexive and symmetric. 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.

Notation 4.1. Let $\Gamma \equiv x_1 : A_1, \dots$ be a pre-context. Let $Dom(\Gamma) =_{\text{def}} \{x_1, \dots\}$ be the set of variables Γ declares, $\Gamma|_{x_i} =_{\text{def}} x_1 : A_1, \dots, x_{i-1} : A_{i-1}$ be the restriction of Γ up to x_{i-1}. Define $\Gamma(x_i) =_{\text{def}} A_i$, viewing Γ as a partial mapping $\Gamma : \mathbf{V} \rightharpoonup \mathbf{T}$. 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. 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 $\mathbf{V}$. Substitution is extended to contexts and judgements componentwise, *e.g.,* if $\Gamma \equiv x_1 : A_1, \dots$ then $\Gamma[N/x] \equiv x_1 : A_1[N/x], x_2 :, \dots$.

We first prove by induction on derivations that the usual good properties for subtyping systems hold: context formation, renaming, thinning, substition and

[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 $y : \Pi x{:}A.\ \mathsf{Power}(y\, x)$.

bound narrowing (replacing $x : A$ with $x : A'$ where $A' : \mathsf{Power}(A)$). Next we show the important formation and type correctness properties.

Proposition 4.2 (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. $\Gamma \rhd \mathsf{Power}(A) : C \quad\Longrightarrow\quad \Gamma \rhd A\ \mathsf{type}.$

Proposition 4.3 (Type correctness).

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

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. For details, see [2]. An important intermediate stage is proving the transitivity of type equality, using this rule:

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

This 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. Unfortunately it seems difficult to prove for λ_{Power}. The key is a *generation principle*, which gives a way of decomposing derivations by stating how a particular judgement was derived. Proposition 4.2 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: if $\Gamma \rhd \lambda x{:}A.\, M : C$ then $C = \Pi x{:}A'.\, B'$, 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

[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.

(λ) there can be several subsumptions and conversions through which $\Pi x{:}A.\,B$ mutates into C:

$$
\begin{array}{c}
\dfrac{x : A \rhd M : B}{\lambda x{:}A.\,M : \Pi x{:}A.\,B} \\
\vdots \\
\dfrac{\lambda x{:}A.\,M : C_j \qquad C_j \le C_{j+1}}{\lambda x{:}A.\,M : C_{j+1}}\ (\textsc{sub}) \\
\vdots \\
\dfrac{\lambda x{:}A.\,M : C_k \qquad\qquad C_k = C_{k+1}}{\lambda x{:}A.\,M : C_{k+1}}\ (\textsc{conv}) \\
\vdots \\
\lambda x{:}A.\,M : C
\end{array}
$$

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.

The traditional syntactic solution to this problem is to give a *syntax-directed* reformulation of the system, eliminating the cut-like rules. 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. 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.

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 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. The idea of rough type-checking comes from [12], which suggested that rough types could be used to give a 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} [2].

5.1 Rough typing system

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

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

(writing **Ty** as short for $\mathbf{Ty}_{\mathcal{K}}$). I use $\tau, \upsilon, \ldots$ to range over **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.

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:

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

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

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:

$$\Gamma_{\mathrm{PERM}} \rhd \mathit{Perm} \ : \ \mathit{int} \to \mathit{Power}(\mathit{int} \to \mathit{int})$$

because, for starters, *Perm* is not defined on all of *int*. In Proposition 5.4, we prove that typability in the full calculus guarantees rough typability. The above example shows that the converse fails, since:

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

but $\mathit{Perm}\, i$ cannot be typed in the full system[7].

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. First, we have the usual thinning, substitution and also strengthening properties for the rough type system. Then we can prove decidability and subject reduction.

Proposition 5.2 (Properties of rough typing).

1. *If $\Gamma \blacktriangleright M : \tau$, then τ is the unique such rough type.*

[7] to prove this rigorously we need to use a generation principle or model construction.

2. Rough type-checking and rough type-inference are decidable.[8]

Proposition 5.3 (Subject reduction for rough typing). *If $\Gamma \blacktriangleright M : \tau$ and $M \twoheadrightarrow_{\beta\eta} M'$, then $\Gamma \blacktriangleright M' : \tau$ too.*

The agreement property below is the important connection between rough types and typing in full λ_{Power}, claimed at the beginning of this section.

Theorem 5.4 (Agreement of rough typing).

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

6 Semantics

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 $c_{A,B} : [\![A]\!] \to [\![B]\!]$. This is a general setting, but it requires a *coherence* property of the interpretation, to show that different ways of putting coercions into a coercion-free judgement have the same interpretation. The coherence property can be difficult to establish. The other kind of model is a *containment semantics* in which subtyping is interpreted as containment between types: $[\![A]\!] \subseteq [\![B]\!]$. There is no problem of coherence in this case, but there is a difficulty with the rule for subtyping Π-types. In the syntax we have $\mathsf{int} \to \mathsf{int} \leq \mathsf{nat} \to \mathsf{int}$, 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 $\to$ is set-theoretic function space. This is usually solved by interpreting $\mathsf{nat} \to \mathsf{int}$ as the collection of all partial functions defined *at least* on $\mathbf{N}$; then the inclusion $\mathbf{Z} \rightharpoonup \mathbf{Z} \subseteq \mathbf{N} \rightharpoonup \mathbf{Z}$ holds. But then we need a universe of values over which to form this "collection of all partial functions," and this is what leads to an *untyped* value space in containment semantics. Typically, the untyped value space is the domain of a model of the untyped λ-calculus, and the denotation of a term is defined using its type-erasure [10]. But it is a surprising overkill to base a semantics for a calculus as simple as $\lambda_{\leq}$ (the extension of $\lambda^{\to}$ with subtyping) on a model of the untyped λ-calculus which requires a universal domain.

For power types, a containment semantics is natural and is the intended model for ASL+. I shall and 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: $[\![A]\!] \subseteq [\![B]\!] \subseteq [\![\tau]\!]$. 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

[8] assuming we can decide syntactic identity of atomic types, *i.e.,* whether $\kappa \equiv \kappa'$.

universe, instead of a universal domain. The final ingredient is the equational theory of subtyping, where the equality of two terms may depend upon the type at which they are viewed. To deal with this, we use PERs rather than sets. The following sections give an abstract model definition for λ_{Power} based on these ideas, beginning from applicative structures. The reason for an abstract definition is to capture both the intended model and a term model; the term model is unusual for using an *external equality* notion rather than quotients (because of this extensionality is not assumed from the start). Space reasons prevent description of the term model here, see [2] for details.

6.1 Structures

A λ_{Power} applicative structure is similar to a typed-applicative structure for $\lambda^{\rightarrow}$ 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 $\mathsf{App}^{\tau,\upsilon}\ :\ \mathcal{D}^{\tau\Rightarrow\upsilon} \rightarrow \mathcal{D}^{\tau} \rightarrow \mathcal{D}^{\upsilon}$ for each $\tau, \upsilon \in \mathbf{Ty}$. Type annotations τ, υ are sometimes omitted for brevity.* □

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 $f(a)$.

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 has $\mathcal{F}^{\kappa} = C_{\kappa}$, $\mathcal{F}^{\tau\Rightarrow\upsilon} = \mathcal{F}^{\tau} \rightarrow \mathcal{F}^{\upsilon}$, $\mathcal{F}^{\mathsf{P}(\tau)} = \mathrm{REL}(\mathcal{F}^{\tau})$, $\mathsf{App}(f, m) = f(m)$, and $\mathsf{Const}(\kappa) = R_{\kappa}$. □

In the full hierarchy structure, $\mathcal{F}^{\mathsf{P}(\tau)}$ 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 6.7) is defined over rough types, the type-constructors are not guaranteed to construct PERs.

6.2 Environments and interpretations

For each roughly-typable context Γ, we define a semantic domain $\mathcal{D}^{\Gamma}$ by induction on Γ, setting $\mathcal{D}^{\langle\rangle} = \{\,\star\,\}$ and $\mathcal{D}^{\Gamma,x:A} = \mathcal{D}^{\Gamma} \times \mathcal{D}^{\tau}$, where $\Gamma\ \blacktriangleright\ A :\ \mathsf{P}(\tau)$ and $\{\,\star\,\}$ 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:

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

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 $x \in \mathit{Dom}(\Gamma_1)$. The notation $\eta|_{x_i}$ stands for the restriction of a Γ-environment η to variables declared before x_i, meaning the shorter tuple $\mathit{fst}^{n-i}(\eta)$ where x_i is the ith variable of n declared in Γ.

Unlike a partial function environment, 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 $\lambda^\rightarrow$ in (set-like) CCCs.

Definition 6.4 (λ_{Power} interpretation). *A* λ_{Power} interpretation *in $\mathcal{D}$ consists of a meaning function $[\![\Gamma \blacktriangleright - : \tau]\!]_{\_} : \mathbf{T} \rightharpoonup \mathcal{D}^\Gamma \to \mathcal{D}^\tau$, 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.* □

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 6.4 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.

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 $G \in \mathit{dom}(R) \to \mathrm{REL}(\mathcal{D}^\upsilon)$, we define $\Pi(R,G) \in \mathrm{REL}(\mathcal{D}^{\tau\Rightarrow\upsilon})$, $\mathcal{P}\mathrm{wr}(R) \in \mathrm{PER}(\mathcal{D}^{\mathsf{P}(\tau)})$ by:

$$\begin{array}{lcl} f\ \Pi(R,G)\ g & \text{iff} & \forall a,b.\,(a\ R\ b) \Longrightarrow \mathsf{App}(f,a)\ \ G(a)\ \ \mathsf{App}(g,b). \\ a\ \mathcal{P}\mathrm{wr}(R)\ b & \text{iff} & \mathsf{Rel}(a) = \mathsf{Rel}(b) \in \mathrm{PER}(\mathcal{D}^\tau) \quad \text{and} \quad \mathsf{Rel}(a) \subseteq R. \end{array}$$

Fact 6.5. *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 6.6 (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 all suitable roughly-typable terms. 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 ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$,*

$$\begin{array}{llcl@{\qquad}llcl} \mathsf{CONST} & [\![\kappa]\!]_\eta & = & \mathsf{Const}(\kappa). & \mathsf{VAR} & [\![x]\!]_\eta & = & \eta^\Gamma(x). \\ \mathsf{CONST2} & R_{[\![\kappa]\!]_\eta} & \in & \mathrm{PER}(\mathcal{D}^\kappa). & \mathsf{APP} & [\![M\,N]\!]_\eta & = & \mathsf{App}([\![M]\!]_\eta, [\![N]\!]_\eta). \end{array}$$

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 $R_{[\![\mathit{Power}(C)]\!]_\eta} = \mathcal{P}\mathrm{wr}(R_{[\![C]\!]_\eta})$.
ABS *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* $\mathit{Dom}(\Gamma_1)$ *such that* $\Phi(\Gamma_1) \subseteq \Gamma_2$, *then* $[\![\Phi(\Gamma_1) \blacktriangleright \Phi(M) : \tau]\!]_{\eta_1} = [\![\Gamma_2 \blacktriangleright M : \tau]\!]_{\eta_2}$
SUBST *If* $\Gamma_1 \equiv \Gamma, x : A, \Gamma'$, $\Gamma_2 \equiv \Gamma, \Gamma'[N/x]$ *then* $[\![\Gamma_1 \blacktriangleright M]\!]_{\eta_1} = [\![\Gamma_2 \blacktriangleright M[N/x]]\!]_{\eta_2}$ *provided* ${\eta_1}^{\Gamma_1}(x) = [\![\Gamma \blacktriangleright N]\!]_{\eta_1|_x} \in \mathit{Rel}([\![\Gamma \blacktriangleright A]\!]_{\eta_1|_x})$. □

Axioms CONST, VAR, APP are standard. CONST2 requires that atomic types denote PERs. FAMILY and SUBSET define the extension of the denotation of types of the form $\Pi x{:}A.\,B$ and $\mathit{Power}(C)$. ABS ensures the soundness of the three equality rules which mention the λ-constructor.

Example 6.7 (Full hierarchy model). We define an interpretation by:

$$\begin{aligned}
\mathit{Rel}(A) &= A \\
[\![\Gamma \blacktriangleright x : \tau \;]\!]_\eta &= \eta^\Gamma(x) \\
[\![\Gamma \blacktriangleright \kappa : \mathsf{P}(\kappa)]\!]_\eta &= R_\kappa \\
[\![\Gamma \blacktriangleright \lambda x{:}A.\,M : \tau \Rightarrow \upsilon]\!]_\eta &= a \mapsto [\![\Gamma, x : A \blacktriangleright M : \upsilon]\!]_{\langle\eta,\,a\rangle} \\
[\![\Gamma \blacktriangleright M\,N : \upsilon]\!]_\eta &= \mathit{App}([\![\Gamma \blacktriangleright M : \tau \Rightarrow \upsilon]\!]_\eta, [\![\Gamma \blacktriangleright N : \tau]\!]_\eta) \\
[\![\Gamma \blacktriangleright \Pi x{:}A.\,B : \mathsf{P}(\tau \Rightarrow \upsilon)]\!]_\eta &= \Pi(R_{[\![A]\!]_\eta}, a \mapsto R_{[\![B]\!]_{\langle\eta,\,a\rangle}}) \\
[\![\Gamma \blacktriangleright \mathit{Power}(A) : \mathsf{P}(\mathsf{P}(\tau))]\!]_\eta &= \mathcal{P}\mathrm{wr}(R_{[\![A]\!]_\eta}) \qquad \square
\end{aligned}$$

Lemma 6.8. *The interpretation defined in Example 6.7 is a model of* λ_{Power}.

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.

6.4 Soundness

Here we show that 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, $\mathit{Rel}([\![A]\!]_\eta)$ 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 6.9. *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 6.10 (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, by $[\![\langle\rangle]\!] = \{\, (\star,\star) \,\}$ and $[\![\Gamma',\, x : A]\!] = \Sigma([\![\Gamma']\!], \eta \mapsto \;\; R_{[\![\Gamma' \blacktriangleright A :\, \mathsf{P}(\tau)]\!]_\eta})$. We say $\eta_1, \eta_2 \in \mathcal{D}^{\Gamma}$ are* related environments satisfying *Γ iff $\eta_1 \;\; [\![\Gamma]\!] \;\; \eta_2$.*

Fact 6.11. *If Γ is roughly-typable and $\eta_1 \;\; [\![\Gamma]\!] \;\; \eta_2$, then for all $x \in Dom(\Gamma)$, $\eta_1^{\Gamma}(x) \quad R_{[\![\Gamma|_x \blacktriangleright \Gamma(x) :\, \mathsf{P}(\tau)]\!]_{\eta_1|_x}} \quad \eta_2^{\Gamma}(x)$.*

Lemma 6.12. *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 in any model:*

WEAK-EXT *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*
$\forall d, e. \quad d \;\; R_{[\![A]\!]_\eta} \;\; e \implies \mathsf{App}([\![\lambda x{:}A.\, M]\!]_\eta, d) \;\; R_{[\![B]\!]_{\langle \eta,\, d\rangle}} \;\; \mathsf{App}([\![\lambda x{:}A.\, N]\!]_\eta, e)$

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

Theorem 6.13 (Soundness for models).

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

Corollary 6.14 (Soundness of Typing). *If $\Gamma \rhd M : A$, then for all η, $\eta \in dom\, [\![\Gamma]\!] \implies [\![M]\!]_\eta \in R_{[\![A]\!]_\eta}$.*

7 Conclusions

This paper introduces the type system λ_{Power}, a predicative fragment of Cardelli's original power type system [4]. 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.

Several important results are established. 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+ [12], from which λ_{Power} grew. The work here first appeared in my thesis supervised by Sannella [1]. 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 (some questions are addressed only in [2] due to lack of space here).

References

[1] David Aspinall. *Type Systems for Modular Programs and Specification.* PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[2] David Aspinall. Subtyping with power types. Draft full version. LFCS, University of Edinburgh, 2000.

[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.

[4] 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.

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

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

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

[8] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.

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

[11] 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.

[12] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

$$\frac{\triangleright \Gamma}{\Gamma \triangleright \kappa : \mathsf{Power}(\kappa)} \qquad (\textsc{atomic})$$

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

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

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

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

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

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

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

$$\frac{\Gamma \triangleright A : \mathsf{Power}(B)}{\Gamma \triangleright \mathsf{Power}(A) : \mathsf{Power}(\mathsf{Power}(B))} \qquad (\mathsf{Power})$$

Fig. 1. Typing rules

$$\frac{}{\triangleright \langle\rangle} \qquad (\textsc{empty})$$

$$\frac{\triangleright \Gamma \qquad \Gamma \triangleright A : \mathsf{Power}(B)}{\triangleright \Gamma, x : A} \qquad (\textsc{extend})$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright M = M : A} \qquad (\textsc{eq-refl})$$

$$\frac{\Gamma \triangleright N = M : A}{\Gamma \triangleright M = N : A} \qquad (\textsc{eq-sym})$$

$$\frac{\Gamma \triangleright M = N : A \qquad \Gamma \triangleright N = P : A}{\Gamma \triangleright M = P : A} \qquad (\textsc{eq-trans})$$

$$\frac{\Gamma, x : A \triangleright M = M' : B}{\Gamma \triangleright \lambda x{:}A.\, M = \lambda x{:}A.\, M' : \Pi x{:}A.\, B} \qquad (\textsc{eq-}\lambda)$$

$$\frac{\begin{array}{l} \Gamma \triangleright M = M' : \Pi x{:}A.\, B \\ \Gamma \triangleright N = N' : A \end{array}}{\Gamma \triangleright M\, N = M'\, N' : B[N/x]} \qquad (\textsc{eq-app})$$

$$\frac{\Gamma, x : A \triangleright M : B \qquad \Gamma \triangleright N : A}{\Gamma \triangleright (\lambda x{:}A.\, M)\, N = M[N/x] : B[N/x]} \qquad (\textsc{eq-}\beta)$$

$$\frac{\Gamma \triangleright M : \Pi x{:}A.\, B}{\Gamma \triangleright \lambda x{:}A.\, M\, x = M : \Pi x{:}A.\, B} \qquad (\textsc{eq-}\eta)$$

Fig. 2. Context and equality rules

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

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

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

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

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

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

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

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

Fig. 3. Rough typing rules