Type Systems for Modular Programs and Specifications

David R. Aspinall

Doctor of Philosophy
University of Edinburgh
1997

Short Contents

Abstract

This thesis studies the foundations of formal program development. It brings together aspects of algebraic specification and type theory and applies them to powerful new mechanisms for modular programming and specification.

The language **ASL+** is the vehicle for the study. It is a typed λ-calculus built on top of a core-level programming language and the algebraic specification language ASL. The λ-calculus expresses the structure of programs and specifications in the same language, allowing higher-order parameterisation of each, and allowing specification of parameterised programs.

ASL+ has a model-theoretic semantics based on an arbitrary institution, and two formal systems: a type-checking system to check well-formedness, and a proof system to prove that a program satisfies a specification or that one specification refines another. The type-checking system builds on simply typed λ-calculus. The proof system is richer: a type theory with subtyping, dependent products, power types, and singleton types. This is a novel combination; establishing even basic properties poses a challenge.

I demonstrate the use of ASL+ with an example program development, introducing some rules and issues such as *sharing*. The formal study begins with two fundamental investigations into sub-languages of ASL+, new typed λ-calculi in their own right. The first calculus $\lambda_{\leq\{\}}$ features singleton types, the second calculus λ_{Power} features power types. Both calculi mix subtyping with type-dependency. I prove that each is strongly normalizing, and has expected admissible rules; for example, $\lambda_{\leq\{\}}$ has subject reduction and minimal typing. The calculus λ_{Power} is given a restricted system for *rough type-checking* which is decidable. Rough types help organize a model definition.

I examine two versions of ASL+ itself. The first is an abstract kernel language which treats the underlying core-level languages as sets of combinators. It is built on a calculus $\lambda_{\mathsf{ASL+}}$ which extends $\lambda_{\leq\{\}}$ and λ_{Power}. Practical examples must be translated into this version of ASL+, because it does not automatically express the sharing behaviour of parameterised programs. Instead of a translation, I give a second version of ASL+ in a specific institution $\mathcal{FPC}$. The institution is based on FPC, a functional language with recursive types, together with an LCF-style extension to higher-order logic. This concrete version of ASL+ has a more powerful type-checking system than the abstract version, so programs and specifications can be written directly without translation.

Preface

Acknowledgements

I'm grateful to many kind people for their help and encouragement along the way. This is a quick summary.

First to my supervisors, Don Sannella, Benjamin Pierce, and (an occasional unofficial supervisor) Andrzej Tarlecki, for their time spent coaching me. Next, to the many friends and colleagues I met at Edinburgh, for discussions technical and not. Special thanks to Claudio Russo, for talks on the subject of Chapter 7 and help implementing prototype type-checkers, and to Adriana Compagnoni for our enjoyable collaboration on subtyping. Also to Healfdene Goguen, Martin Hofmann, Neil Ghani, Christoph Lüth, Chris Owens, James McKinna, and Andrew Wilson. Thanks to Luca Cardelli for listening to my explanation of my work on power types. Thanks to Andrew Kennedy for reading a draft at the last minute, and to everyone else who read parts of drafts: Adriana Compagnoni, Healf Goguen, Christoph Lüth, Markus Wenzel. Thanks to Larry Paulson and Graham Titmus for guidance early in my career.

I'm grateful to my examiners, Martin Wirsing and Rod Burstall, for their suggestions and their time spent examining this thesis.

Finally, to my friends and family who have supported me over the past years. This thesis is dedicated to my parents, Roy and Eurwen, and my grandmother Mary Wilkinson.

Edinburgh,
August 1997.

Contents

Figures

1 Introduction

This thesis studies modular specification and programming, combining aspects of algebraic specification and type theory. This introduction provides some background, including fundamental definitions.

The vehicle used for the study is a language called **ASL+** which follows the algebraic tradition. It has a formal, model-theoretic semantics which is paramount. The language and its semantics are based on an arbitrary *institution* which provides the building blocks for constructing programs and specifications.

TO DEVELOP PROGRAMS FORMALLY, we need both a specification language and a programming language, and a formal semantics for each of them. The syntax of the two languages must be connected, so specifications can describe properties of programs. The semantics must be connected, so the logic of the specification language indeed expresses the behaviour of programs.

Whatever way these connections are made, there is a fundamental question which we can ask, and at least some of the time, which we can answer. The question is: does the program P implement, or *satisfy*, the specification SP? The statement that P satisfies SP is written

$$P : SP.$$

Another question, aiming towards a gradual development of specifications towards programs, is: does one specification SP_2 implement, or *refine*, another specification SP_1? This is written as

$$SP_1 \rightsquigarrow SP_2.$$

Intuitively, one specification refines to another when some design or coding decisions are taken, reducing the collection of programs that count as correct implementations.

In algebraic specification, programs are modelled by *algebras*, and we write $[\![P]\!]$ for the algebra denoted by the program P. The most direct interpretation of a specification SP is the collection of algebras which model programs that implement SP. An algebra that models an implementation of SP is said to be a *model of SP*, and we write $[\![SP]\!]$ for the class of models of SP. If $[\![SP]\!] = \emptyset$, then SP cannot be implemented and is said to be *inconsistent*.

A language that has a *model-level semantics* like this follows the model-theoretic approach advocated by Sannella and Tarlecki [1992]. With this approach satisfaction is easily defined:

$$P : SP \qquad \Longleftrightarrow \qquad [\![P]\!] \in [\![SP]\!].$$

Refinement is also easily defined. If $SP_1 \rightsquigarrow SP_2$ then every algebra which is a model of SP_2 must also be a model of SP_1, so

$$SP_1 \rightsquigarrow SP_2 \qquad \Longleftrightarrow \qquad [\![SP_2]\!] \subseteq [\![SP_1]\!]$$

where $\subseteq$ denotes class inclusion.

Refinement is important because proving each development stage correct separately is more tractable than proving correctness of the final program in one go. If one ensures *vertical composition* for refinements, so that $SP_1 \rightsquigarrow SP_2$ and $SP_2 \rightsquigarrow SP_3$ implies $SP_1 \rightsquigarrow SP_3$, then the same end is achieved:

$$\frac{SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \qquad [\![P]\!] \in [\![SP_n]\!]}{[\![P]\!] \in [\![SP_1]\!]}$$

With the definition of the refinement relation as inclusion, vertical composition obviously holds. (The other direction, *horizontal composition*, concerns operations used to put specifications together: it holds for a unary operation f if $SP_1 \rightsquigarrow SP_2$ implies $f(SP_1) \rightsquigarrow f(SP_2)$. With the definitions above, this is true when f is monotonic.)

Three kinds of parameterisation

In this thesis I study the language **ASL+**, which was invented by Sannella, Sokołowski, and Tarlecki, Sannella, Sokołowski, and Tarlecki [1990, 1992], and later christened by Sannella. ASL+ can express both programs and specifications (so it is a *wide-spectrum* language) and it has a model-level semantics following the outline above. But ASL+ has more than just specifications and programs; as well, it can express three different kinds of parameterisation useful for program development: *parameterised programs, parameterised specifications* and *specifications of parameterised programs.*

- A *parameterised program* acts as a function from programs to programs, and is used for building large programs in a modular fashion. The advantages of reusability and separate development are well-known. Several modern programming languages have module systems which can express parameterised programs; in the module system of SML (Standard ML, Paulson [1991]), for example, parameterised programs are known as *functors.*

  In ASL+, parameterised programs are written using λ-abstractions.

- A *parameterised specification* acts as a function from specifications to specifications, and is used for building large specifications. Many specification languages have some form of parameterised specification; an example is ASL Sannella and Wirsing [1983]. But not all forms of parameterisation found in specification languages act as functions from specifications to specifications.

  In ASL+, parameterised specifications are also written using λ-abstractions.

- A *specification of a parameterised program* acts as a collection of functions from programs to programs, allowing the modular structure of a program to be specified. In EML (Extended ML, Kahrs et al. [1994]), *functor specifications* are used to specify Standard ML functors, so they are specifications of parameterised programs.

  In ASL+, specifications of parameterised programs are written using Π-abstractions.

The third kind of parameterisation is one of the novelties in ASL+; specifications of parameterised programs are sometimes called *Π-specifications.* Sannella, Sokołowski, and Tarlecki introduced Π-specifications when they realised that the functor specifications of Extended ML could not be expressed in ASL, which they hoped to use as a semantic kernel.

Sometimes the structure of an implementation mirrors the structure of its requirement specification, but there is no reason why this should be enforced by the language. It may be easy to express the specification using a decomposition which is unnatural or even impossible from a programming point of view. Researchers argue that the structure of a requirements specification should not determine the structure of an implementation [Fitzgerald and Jones, 1990, Sannella et al., 1992], although some languages and development methodologies *do* enforce this. (For good reason — if the implementation has the same decomposition as the specification, it is liable to be easier to prove correct.) Adding Π-specifications to a language allows the structure of the implementation to enter the realm of specification, so the most flexible approach is to have both Π-abstracted and λ-abstracted specifications.

Much more can be said about methodology and program structure versus specification structure. Sannella et al. [1992] give discussion and references, and also some analysis which characterises when it is *possible* to implement a parameterised specification with a parameterised program, decomposing the implementation in the same way as the specification. (We might even consider adding a special operator to the language of ASL+ to correspond to this development step.) For this thesis, the issues are explored a little bit in the example formal development in Chapter 2, which motivates another kind of parameterisation: specifications parameterised upon programs. Mainly I am concerned with semantic foundations rather than methodology, and the main topic is the type systems underlying ASL+.

Two kinds of type system, one with subtyping

ASL+ is a typed λ-calculus, with both λ-abstraction and Π-abstraction. The forms of abstraction are entirely general, and ASL+ has higher-order versions of all the forms of parameterisation mentioned. Moreover, the language contains a novel term former $\mathit{Spec}(SP)$ which is used to parameterise on refinements of SP.

Not surprisingly, this makes for a complicated type-checking system. In fact, we study two kinds of type system, one for *type-checking*, and the other for *satisfaction.*

- A *type-checking system* proves the structural well-formedness of syntactic expressions. In general, only typed terms have a denotation in the semantics, and we aim for type-checking to be decidable.

- A *satisfaction system* proves the implementation relations: that a program satisfies a specification, or that one specification refines another. Showing satisfaction involves proving logical consequences, to show that a program satisfies some axioms, for example. Depending on the logic used, satisfaction is often undecidable.

We should only prove satisfaction between typable objects, so satisfaction subsumes type-checking. But it is useful to separate the type-checking component of satisfaction, so that type-checkers can be used to check source texts mechanically, just as with ordinary programming languages. The type-checking part of satisfaction is a fairly obvious fragment which should be decidable.[1] An important topic not addressed in this thesis is the study of other decidable fragments of satisfaction systems, and strategies for proof search.

Both type-checking and satisfaction systems are theories of typing judgements; the difference is what the types are. In a satisfaction system, types are specifications, but in a type-checking system the types have a simpler structure. To begin with, I study the typed λ-calculus underlying ASL+ in a pure setting, and type-checking is called *rough typing* to distinguish it from typing in the system with "full" types. When the pure theory is applied to ASL+, the system with full types is the one used for proving satisfaction.

Satisfaction systems are based on type systems with subtyping. The correspondence is easy to see:

$$\begin{array}{ccc} \textit{elementhood},\ P : A & \iff & \textit{satisfaction},\ P : A \\ \textit{subtyping},\ A \leq B & \iff & \textit{refinement},\ B \rightsquigarrow A \end{array}$$

The *elementhood* of a term P in a type A is like *satisfaction* of a specification A by a program P. Saying that a type A is a *subtype* of another type B is like saying that the specification A is a *refinement* of the specification B.

Of course, the viewpoint of types-as-specifications is not new; it is pursued in many variants of Martin-Löf's type theory [Nordström et al., 1990] and in studies influenced by algebraic specification [Streicher and Wirsing, 1991, Luo, 1993]. What differs here is that programs and specifications are built not in a type theory directly, but using other languages. The typed λ-calculus of ASL+ is a structuring mechanism for programming and specification in-the-large. Thus we extend Burstall and Lampson's original conception of *modular programming as typed functional programming* [Burstall, 1984, Lampson and Burstall, 1988] to the realms of specification.

The rest of this introduction is as follows. Section 1.1 introduces the notion of *institution*, which provides an abstract model theory for specifications and programs. ASL+ is based on an arbitrary institution. Section 1.2 describes ASL, the specification language upon which ASL+ is based. Section 1.3 describes the limited parameterisation capabilities of ASL, and Section 1.4

[1] Some researchers fancy programming with powerful *semi-decidable* type-checking systems, but here I am talking about type-checking in-the-large, for putting together program *modules*, which is usually simpler.

then contains a brief overview of ASL+ and its vastly extended parameterisation features. Section 1.5 mentions some conventions and notations used in the thesis and Section 1.6 gives an outline of the thesis content.

1.1 Institutions

Classically, algebraic specification models programs as many-sorted algebras, and expresses their properties using many-sorted equational logic. This abstracts away from the details that are not concerned with correctness; algorithmic complexity, for example, plays no part in classical algebraic specification.

It was soon realised that many-sorted algebras are not flexible enough to handle all of the data that programs might manipulate (e.g., higher-order functions), and equational logic is not expressive enough to describe some properties of the data (e.g., exception conditions) in a natural way, if at all.

Researchers have proposed many variants of algebra and logic to solve these problems, yet it seems that much of the study of specification languages can be abstracted away from these "low-level" details. A tool for this is the theory of *institutions*, which provides an *abstract model theory*, a standard way of viewing an algebra and its related logic. Unlike model theory in logic, the theory of institutions allows us to *change notation*, by relating models over different signatures. This is essential for applied logics in computer science, where program parts change name according to scope.

Definition 1.1 (Institution [Goguen and Burstall, 1992]).
An *institution* $\mathcal{I}$ consists of:

- A category of signatures, $\mathbf{Sign}^{\mathcal{I}}$.
  We often use Σ to range over the objects of this category, and $\sigma : \Sigma \to \Sigma'$ to range over the morphisms. Intuitively, a signature provides the vocabulary for expressing properties of models.

- A sentence functor $\mathbf{Sen}^{\mathcal{I}} : \mathbf{Sign}^{\mathcal{I}} \to \mathbf{Set}$.
  For each signature Σ, there is a set of Σ-sentences in the logic, $\mathbf{Sen}^{\mathcal{I}}(\Sigma)$; for each signature morphism $\sigma : \Sigma \to \Sigma'$ there is a translation of sentences $\mathbf{Sen}^{\mathcal{I}}(\sigma)$.

- A model functor $\mathbf{Mod}^{\mathcal{I}} : \mathbf{Sign}^{\mathcal{I}} \to \mathbf{Cat}^{op}$.
  For each signature Σ, there is a category of *Σ-models*, $\mathbf{Mod}^{\mathcal{I}}(\Sigma)$ and for each signature morphism $\sigma : \Sigma \to \Sigma'$ there is a functor $\mathbf{Mod}^{\mathcal{I}}(\sigma)$, known as the *$\sigma$-reduct functor*, which translates Σ'-models (and Σ'-model morphisms) to Σ-models (and Σ-model morphisms).

- For every signature Σ, a *satisfaction relation* $\models^{\mathcal{I}}_{\Sigma} \subseteq |\mathbf{Mod}^{\mathcal{I}}(\Sigma)| \times \mathbf{Sen}^{\mathcal{I}}(\Sigma)$ between Σ-models and Σ-sentences.
  The satisfaction relation provides us with the *truth* of a sentence in a particular model.[2]

For any signature morphism $\sigma : \Sigma \to \Sigma'$, sentence $\varphi \in \mathbf{Sen}^{\mathcal{I}}(\Sigma)$ and model $m' \in |\mathbf{Mod}^{\mathcal{I}}(\Sigma')|$, the *satisfaction condition*

$$m' \models^{\mathcal{I}}_{\Sigma'} \mathbf{Sen}^{\mathcal{I}}(\sigma)(\varphi) \iff \mathbf{Mod}^{\mathcal{I}}(\sigma)(m') \models^{\mathcal{I}}_{\Sigma} \varphi$$

must hold. □

The satisfaction condition states that satisfaction is preserved when translating models and sentences along a signature morphism, hence the slogan "truth is invariant under change of notation."

Notation for institutions. When working with a particular institution, I often drop the $\mathcal{I}$ superscripts. I often call models in an arbitrary institution "algebras" to avoid overloading the word "model" too much. The function $\mathbf{Sen}(\sigma)$ is written simply as σ. For a category $\mathbf{C}$ such as $\mathbf{Sign}$ or $\mathbf{Mod}(\Sigma)$, sometimes I write $\mathbf{C}$ to mean the objects $|\mathbf{C}|$ of $\mathbf{C}$. The reduct functor $\mathbf{Mod}(\sigma)$ is usually used as a function on models; this function is abbreviated $-|_{\sigma}$, or when we have a signature inclusion $\iota : \Sigma \hookrightarrow \Sigma'$, as $-|_{\Sigma}$, the function which returns the *restriction* of a Σ'-model to a Σ-model. If $m = m'|_{\sigma}$ then we say that m' is a *σ-expansion* of m. For the satisfaction relation, if $\Phi \subseteq \mathbf{Sen}(\Sigma)$ is a set of sentences then $m \models_{\Sigma} \Phi$ holds iff for all $\varphi \in \Phi$, $m \models_{\Sigma} \varphi$; if $k \subseteq \mathbf{Mod}(\Sigma)$ is a collection of models, then $k \models_{\Sigma} \varphi$ iff for all $m \in k$, $m \models_{\Sigma} \varphi$. Semantic entailment between sentences is defined in the usual way: $\Phi \models_{\Sigma} \varphi$ iff for all m, $m \models_{\Sigma} \Phi$ implies $m \models_{\Sigma} \varphi$. Finally, $[\![\Phi]\!]$ denotes the class of Σ-models which satisfy Φ.

A familiar case of Definition 1.1 is the institution $\mathcal{FOL}$ of first-order logic.

Example 1.2 (First-order logic $\mathcal{FOL}$). A *first order signature* is a pair $\Sigma = \langle \Omega, \Upsilon \rangle$ where for each $n \in \omega$, Ω_n is a set of function symbols of arity n and Υ_n is a set of relation symbols of arity n. A morphism of signatures $\sigma : \Sigma \to \Sigma'$ is a pair of mappings $\sigma_f : \Omega \to \Omega'$ and $\sigma_r : \Upsilon \to \Upsilon'$.

A Σ-sentence is the usual notion of a first-order sentence, i.e., a formula φ built inductively by $R^n(m_1, \ldots, m_n)$, $\neg\varphi$, $\varphi_1 \bigwedge \varphi_2$, $\varphi_1 \bigvee \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, $\forall x.\varphi_1(x)$ and $\exists x.\varphi_1(x)$, where the m_i are terms built from variables and constants in Ω and φ_1, φ_2 are formale. The translation of a sentence under

[2]The fine-scale satisfaction relation of an institution generates the implementation relations for programs and specifications, also called "satisfaction relations" in this thesis.

a signature morphism $\sigma = (\sigma_f, \sigma_r)$ is the homomorphic extension of the renamings of the function symbols σ_f and the relation symbols σ_r.

A Σ-model m is a first-order model, i.e., a set $|m|$ together with assignments of the function symbols $f \in \Omega_n$ to n-ary mappings f_m on $|m|$ and assignments of the relation symbols $R \in \Upsilon_n$ to n-ary relations R_m on m. A Σ-model morphism is a homomorphism between Σ-models. The translation of Σ'-model m to a Σ-model $m|_\sigma$ along a signature morphism $\sigma : \Sigma \to \Sigma'$ is defined by $f_{m|_\sigma} = \sigma(f)_m$ and $R_{m|_\sigma} = \sigma(R)_m$.

Satisfaction $M \models \varphi$ is defined via the usual notion of the truth of φ for all valuations of free variables into $|m|$. □

Another common example is the institution of equational logic, $\mathcal{EQ}$. A signature in $\mathcal{EQ}$ is a many-sorted algebraic signature; terms over a signature are built up from the function symbols and a set of variables, and sentences are universally-quantified equations between two terms of the same sort. Models are many-sorted algebras, and satisfaction is defined in the obvious way. This and other examples of institutions are spelt out by Goguen and Burstall [1992].

1.1.1 Institutional semantics

An institution provides some of the semantic foundation for formal program development. An *institutional semantics* uses parts of an institution to interpret a programming and specification language in a standard way.

A program P determines a signature $Sig(P)$ which describes its vocabulary: typically, the names and types of its variable and procedure declarations. If $Sig(P) = \Sigma$, we say that P is a Σ-program. The denotation of a Σ-program is a Σ-algebra, so $[\![P]\!] \in \mathbf{Mod}(\Sigma)$. Similarly, a specification SP determines a signature $Sig(SP)$, and if $Sig(SP) = \Sigma$, we call SP a Σ-specification. The denotation of a Σ-specification is a class of models over Σ, so $[\![SP]\!] \subseteq \mathbf{Mod}(\Sigma)$. The semantics of programs and specifications are connected by the same notion of signature and model.

Sentences are used in the syntax of specifications to express logical properties. Programs, on the other hand, are usually written in a more restricted (executable) language. For example, orthogonal term rewriting systems [Dershowitz and Jouannaud, 1990] are a suitable programming language for $\mathcal{EQ}$; programs are special sets of oriented equations which can only be combined in certain ways. In a different setting, orthogonal term rewriting systems could be used as executable specifications for a more low-level programming language. To contrast, in an institution for constructive type theory, programs are not more restricted than specifications because both are terms from the same language — but with different types [Luo, 1993]. The framework of institutional semantics encompasses a wide variety of settings, but whatever the setting, the syntax of programs and the syntax of specifications are connected by the sentences of the institution.

1.1.2 Proof systems for institutions

To develop programs formally using an institutional semantics, we need proof systems for deriving instances of the two implementation relations, $P : SP$ and $SP_1 \rightsquigarrow SP_2$. Because sentences appear in specifications, proving either of these relations can involve logical reasoning, so we need a proof system for reasoning in the logic of the institution. Logically, the important thing about a proof system is the consequence (derivability) relation which it gives rise to.

Definition 1.3 (Consequence Relation).
A *consequence relation* $\vdash$ consists of:

- A category of signatures, $\mathbf{Sign}^{\vdash}$.
- A sentence functor $\mathbf{Sen}^{\vdash} : \mathbf{Sign}^{\vdash} \to \mathbf{Set}$.
- For each $\Sigma \in \mathbf{Sign}^{\vdash}$, a relation $\vdash_\Sigma \subseteq Pow(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$.

As usual, $s_1, \ldots, s_n \vdash_\Sigma s$ means $(\{ s_1, \ldots, s_n \}, s) \in \vdash_\Sigma$. Each relation $\vdash_\Sigma$ must satisfy the following structural properties:

1. *Reflexivity.* $s \vdash_\Sigma s$.
2. *Transitivity.* If $\Phi \vdash_\Sigma \varphi$ and $\Phi, \varphi \vdash_\Sigma \varphi'$ then $\Phi \vdash_\Sigma \varphi'$.
3. *Weakening.* If $\Phi \vdash_\Sigma \varphi$ then $\Phi, \varphi' \vdash_\Sigma \varphi$.
4. *Translation.* If $\Phi \vdash_\Sigma \varphi$ then $\sigma(\Phi) \vdash_{\Sigma'} \sigma(\varphi)$.

where $\Phi \subseteq \mathbf{Sen}(\Sigma)$, $\varphi, \varphi' \in \mathbf{Sen}(\Phi)$, and $\sigma : \Sigma \to \Sigma'$. □

This definition adds signatures to the usual definition of a consequence relation [Avron, 1991]; the structural properties are the usual ones, together with the condition that consequence is preserved by signature morphisms.

Fact 1.4. The satisfaction relation $\models^{\mathcal{I}}$ of an institution $\mathcal{I}$ extends to a consequence relation (with the definition of $\Phi \models^{\mathcal{I}} \varphi$ given earlier).

Definition 1.5 (Deduction relation for an institution).
A *deduction relation* for an institution $\mathcal{I}$ is a consequence relation $\vdash^{\mathcal{I}}$ for $\mathbf{Sign}^{\mathcal{I}}$ and $\mathbf{Sen}^{\mathcal{I}}$, which is *sound* for $\models^{\mathcal{I}}$, so that

$$\Phi \vdash^{\mathcal{I}}_{\Sigma} \varphi \quad \Longrightarrow \quad \Phi \models^{\mathcal{I}}_{\Sigma} \varphi$$

for all $\Sigma \in \mathbf{Sign}^{\mathcal{I}}$, $\Phi \subseteq \mathbf{Sen}^{\mathcal{I}}(\Sigma)$, and $\varphi \in \mathbf{Sen}^{\mathcal{I}}(\Sigma)$.
A deduction relation is *complete* if the opposite implication also holds.

Several researchers have studied institutions extended with a deduction relation; Meseguer [1989] for example, gives almost the same definitions as above. He calls a consequence relation indexed by signatures an *entailment system*, and he calls the combination of an institution and a sound deduction relation a *logic*.

As well as a deduction relation for the logic of the institution, we may want sound proof systems for proving properties of programs and specifications. Such proof systems give rise to relations $P \vdash \varphi$, which implies $[\![P]\!] \models_{Sig(P)} \varphi$, and $SP \vdash \varphi$, which implies $[\![SP]\!] \models_{Sig(SP)} \varphi$. These relations are subsumed by $P : SP$ and $SP \leq SP'$ in the language ASL.

1.2 ASL

ASL [Sannella and Wirsing, 1983] is a specification language composed of a small number of *specification building operators.* The operators are low-level, but powerful enough to allow high-level languages to be understood by translating into ASL, using it as a *kernel language.* Sannella and Tarlecki [1988a] give ASL an institution independent treatment, which I shall briefly review.

The syntax of ASL consists of specification expressions *SP*, built from a context-free grammar:

$$\begin{array}{lll} SP & ::= & \Sigma \\ & | & \mathbf{impose}\ \Phi\ \mathbf{on}\ SP \ \mid\ \mathbf{translate}\ SP\ \mathbf{by}\ \sigma \\ & | & \mathbf{derive\ from}\ SP\ \mathbf{by}\ \sigma \ \mid\ SP\ \mathbf{union}\ SP \\ & | & \mathbf{minimal}\ SP\ \mathbf{wrt}\ \sigma \ \mid\ \mathbf{iso\text{-}close}\ SP \\ & | & \mathbf{abstract}\ SP\ \mathbf{wrt}\ \Phi\ \mathbf{via}\ \sigma \end{array}$$

where Σ is a signature from the underlying institution, σ is a signature morphism, and Φ is a set of sentences.

To write specifications with ASL in some particular institution, we need additional institution-*dependent* syntax for writing the "semantic" parts of the specification expression grammar. Concrete syntax for a particular institution will be defined rigorously in Chapter 6, but for most of the thesis I shall use intuitive syntax for writing signatures, signature morphisms, and algebras, in a typical institution. For example,

```
$\Sigma_c =_{def}$           $\Sigma_d =_{def}$          $\sigma_{cd} =_{def}$                  $P_c =_{def}$
sig                sig               $[c \mapsto d, v \mapsto w]$   alg
   type $c$              type $d$                                type $c = nat$
   val $v : c \to c$       type $e$                                val $v = succ$
end                   val $w : d \to d$                      end
                   end
```

This defines two signatures Σ_c and Σ_d, a signature morphism $\sigma_{cd} : \Sigma_c \to \Sigma_d$ between them and a Σ_c-algebra, P_c. This assumes that *nat* stands for an implementation of the natural numbers and *succ* is the successor function on *nat*.

Here is an informal explanation of the ASL operators, in terms of the model classes they define. Formal definitions follow in Definition 1.6.

Σ The class of all Σ-algebras; a trivial specification.

impose Φ **on** *SP*
: The models of *SP* which satisfy the axioms Φ.

translate *SP* **by** σ
: The models which are σ-expansions of models of *SP*. Useful for expanding the signature of a specification with some extra uninterpreted symbols.

derive from SP' **by** σ
: The σ-reducts of the models of SP'. Useful for hiding symbols used in a specification.

SP_1 **union** SP_2
: The union of specifications SP_1 and SP_2, which consists of the intersection of the models of SP_1 and SP_2.

minimal *SP* **wrt** σ
: The models of *SP* which are minimal expansions of their σ-reducts, removing models which are "larger" than necessary. For example, restricting to reachable models.

iso-close *SP*
: The closure of the models of *SP* under isomorphism.

abstract *SP* **wrt** Φ **via** σ

The closure of the models of *SP* under a behavioural equivalence relation determined by a set of sentences Φ and signature morphism σ. Used for abstracting away from particular models to include all models which behave in a similar way.

ASL is a low-level formalism, and we often use combinations of operators when writing specifications. Derived syntax is useful, for example:

- $\langle \Sigma, \Phi \rangle$, which abbreviates **impose** Φ **on** Σ.
- the operator **enrich**, used like this:

  ```
  enrich SP
    with sig
           ...
         end
         axioms Φ
  end
  ```

  which abbreviates the use of translate and union:

  (**translate** *SP* **by** ι) **union** $\langle \Sigma', \Phi \rangle$

  where Σ' is the signature Σ of *SP*, extended by the new declarations in ellipsis, and $\iota : \Sigma \hookrightarrow \Sigma'$ is the inclusion of Σ in Σ'.

In this way, higher-level constructs are quickly created. A fully formal treatment should describe how derived syntax is parsed and type-checked, as well as its translation to ASL operators. Translating well-typed source terms should give terms which can be type-checked in ASL.

1.2.1 Type-checking

To type-check ASL expressions, we use the set of *type-checking rules* shown in Figure 1.1. These determine the signature of a specification expression:

$\blacktriangleright\ SP \implies S(\Sigma)$ *SP* is a Σ-specification.

The rules are deterministic, so a specification has at most one type. This is equivalent to a definition by structural induction on *SP*, and justifies writing $Sig(SP)$ to stand for the Σ such that $\blacktriangleright\ SP \implies S(\Sigma)$.

If concrete syntax is provided for the parts of the ambient institution, the "semantic" premises in Figure 1.1 like $\Phi \subseteq \mathbf{Sen}(\Sigma)$ can be replaced by further type-checking judgements, to check the well-formedness of syntactic representations for sentences, etc. This is carried out for a particular institution in Chapter 6.

$$\frac{}{\blacktriangleright\ \Sigma \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP \;\Rightarrow\; S(\Sigma) \qquad \Phi \subseteq \mathbf{Sen}(\Sigma)}{\blacktriangleright\ \mathbf{impose}\ \Phi\ \mathbf{on}\ \ SP \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP \;\Rightarrow\; S(\Sigma) \qquad \sigma : \Sigma \to \Sigma'}{\blacktriangleright\ \mathbf{translate}\ SP\ \mathbf{by}\ \ \sigma \;\Rightarrow\; S(\Sigma')}$$

$$\frac{\blacktriangleright\ SP' \;\Rightarrow\; S(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\blacktriangleright\ \mathbf{derive\ from}\ \ SP'\ \mathbf{by}\ \ \sigma \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP_1 \;\Rightarrow\; S(\Sigma) \qquad \blacktriangleright\ SP_2 \;\Rightarrow\; S(\Sigma)}{\blacktriangleright\ SP_1\ \mathbf{union}\ SP_2 \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP \;\Rightarrow\; S(\Sigma) \qquad \sigma : \Sigma' \to \Sigma}{\blacktriangleright\ \mathbf{minimal}\ SP\ \mathbf{wrt}\ \sigma \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP \;\Rightarrow\; S(\Sigma)}{\blacktriangleright\ \mathbf{iso\text{-}close}\ \ SP \;\Rightarrow\; S(\Sigma)}$$

$$\frac{\blacktriangleright\ SP \;\Rightarrow\; S(\Sigma) \qquad \sigma : \Sigma \to \Sigma' \qquad \Phi \subseteq \mathbf{Sen}(\Sigma')}{\blacktriangleright\ \mathbf{abstract}\ SP\ \mathbf{wrt}\ \ \Phi\ \mathbf{via}\ \ \sigma \;\Rightarrow\; S(\Sigma)}$$

Figure 1.1: Type-checking ASL operators

1.2.2 Semantics

The semantics of a specification expression *SP* can now be defined conveniently by induction on a typing derivation $\blacktriangleright\ SP \implies S(\Sigma)$. This is equivalent to a definition by induction on the structure of *SP*, but ensures that we only consider well-typed expressions.

Definition 1.6 (Semantics of ASL specifications).
Let $\blacktriangleright\ SP \implies S(\Sigma)$. We define $[\![\blacktriangleright\ SP \implies S(\Sigma)]\!] \in \mathbf{Spec}(\Sigma)$ by induction on the typing derivation of $\blacktriangleright\ SP \implies S(\Sigma)$,

$$[\![\blacktriangleright\ \Sigma \implies S(\Sigma)]\!] = |\mathbf{Mod}(\Sigma)|$$

$$[\![\blacktriangleright\ \mathbf{impose}\ \Phi\ \mathbf{on}\ SP \implies S(\Sigma)]\!] = \\ \{m \in [\![\blacktriangleright\ SP \implies S(\Sigma)]\!] \mid m \models_\Sigma \Phi\}$$

$$[\![\blacktriangleright\ \mathbf{translate}\ SP\ \mathbf{by}\ \sigma \implies S(\Sigma')]\!] = \\ \{m \in \mathbf{Mod}(\Sigma') \mid m|_\sigma \in [\![\blacktriangleright\ SP \implies S(\Sigma)]\!]\}$$

$$[\![\blacktriangleright\ \mathbf{derive\ from}\ SP'\ \mathbf{by}\ \sigma \implies S(\Sigma)]\!] = \\ \{m|_\sigma \mid m \in [\![\blacktriangleright\ SP' \implies S(\Sigma')]\!]\}$$

$$[\![\blacktriangleright\ SP_1\ \mathbf{union}\ SP_2 \implies S(\Sigma)]\!] = \\ [\![\blacktriangleright\ SP_1 \implies S(\Sigma)]\!] \cap [\![\blacktriangleright\ SP_2 \implies S(\Sigma)]\!]$$

$$[\![\blacktriangleright\ \mathbf{minimal}\ SP\ \mathbf{wrt}\ \sigma \implies S(\Sigma)]\!] = \\ \{m \in [\![\blacktriangleright\ SP \implies S(\Sigma)]\!] \mid m \text{ is } \sigma\text{-minimal in } \mathbf{Mod}(\Sigma)\}$$

$$[\![\blacktriangleright\ \mathbf{iso\text{-}close}\ SP \implies S(\Sigma)]\!] = \\ \{m \in \mathbf{Mod}(\Sigma) \mid \exists m' \in [\![\blacktriangleright\ SP \implies S(\Sigma)]\!].\, m \cong m'\}$$

$$[\![\blacktriangleright\ \mathbf{abstract}\ SP\ \mathbf{wrt}\ \Phi\ \mathbf{via}\ \sigma \implies S(\Sigma)]\!] = \\ \{m \in \mathbf{Mod}(\Sigma) \mid \exists m' \in [\![\blacktriangleright\ SP \implies S(\Sigma)]\!].\, m \equiv_\sigma^\Phi m'\} \qquad \square$$

A couple of notations used above need explanation.

An algebra $m \in \mathbf{Mod}(\Sigma)$ is *σ-minimal* in a collection $K \subseteq \mathbf{Mod}(\Sigma)$ if $m \in K$ and m has no proper subalgebra in K with an isomorphic σ-reduct. This requires a notion of subalgebra, which can be provided by giving a factorisation system on each category $\mathbf{Mod}(\Sigma)$; subalgebras are then the subobjects of the factorisation system. Minimality requirements can be used to express reachability; informally, reachable models are ones in which certain induction principles are valid.

Two models $m_1, m_2 \in \mathbf{Mod}(\Sigma)$ are *observationally equivalent wrt Φ via σ*, written $m_1 \equiv_\sigma^\Phi m_2$, iff for every σ-expansion m_1' of m_1 there is a σ-expansion m_2' of m_2 such that

$$\forall \varphi \in \Phi.\ \ m_1' \models_{\Sigma'} \varphi \iff m_2' \models_{\Sigma'} \varphi,$$

and vice-versa: for every σ-expansion m_2' of m_2, there is a m_1', and so on. Intuitively, this captures a model-theoretic notion of observational equivalence because Σ' can be a larger signature than Σ, introducing some symbols for "free variables." Then a σ-expansion of a Σ-model corresponds to an environment for the free variables. The sentences Φ are the permissible observations.

For more details of this institution-independent semantics of ASL, see Sannella and Tarlecki [1988a].

1.2.3 Satisfaction in ASL

Now that we have a type-checking system for ASL and a semantics based on it, we can define a satisfaction system for proving refinement of specifications $SP \le SP'$, or, when programs are added, satisfaction of specifications $P : SP$.

The *soundness* of the satisfaction system is vital. To even consider it, we should have the important *agreement* connection between the type-checking system and the satisfaction system; it only makes sense to assert $SP \le SP'$ (or $P : SP$) if the signatures of SP and SP' (or P and SP) are the same. For refinement, we expect that whenever $SP \le SP'$ is provable, then there is some Σ such that $\blacktriangleright\ SP \implies S(\Sigma)$ and $\blacktriangleright\ SP' \implies S(\Sigma)$.

As mentioned above, a satisfaction system for ASL also gives proof relations for proving properties of programs and specifications. We can define:

$$P \vdash \varphi \quad =_{\text{def}} \quad P : \textbf{impose}\ \{\ \varphi\ \}\ \textbf{on}\ \ Sig(P)$$

$$SP \vdash \varphi \quad =_{\text{def}} \quad SP \le \textbf{impose}\ \{\ \varphi\ \}\ \textbf{on}\ \ Sig(SP)$$

However, the definition of both $P : SP$ and $SP \le SP'$ must refer to the satisfaction relation of the institution to handle **impose** specifications in the first place, so it may be more realistic to give proof systems for the relations $P \vdash \varphi$ and $SP \vdash \varphi$ first, then define $P : SP$ and $SP \le SP'$ using them.

I shall not give the rules for a satisfaction system here; example rules of satisfaction systems for ASL will be given in Chapters 5 and 7. There is still scope for further research into finding good systems for proving satisfaction for ASL, but the main concern of this thesis is how to extend such a system to ASL+.

Pattern of development

The pattern followed above for ASL is typical. First we define the context-free syntax of a language, followed by some type-checking rules to restrict to meaningful expressions. Then we define a semantics of well-typed terms by induction over typing derivations. Then a satisfaction system can be defined, which is a proof system for the two implementation relations. The satisfaction system must be proven sound with respect to the semantics, and it must be shown to agree with the type-checking system. This pattern of development, or some part of it, will be repeated several times over in the rest of the thesis.

In general both the type-checking rules and the satisfaction rules are context-sensitive, as when parameterisation is added.

1.3 Parameterisation in ASL

ASL has a limited mechanism for writing parameterised specifications using a λ-calculus notation. The limitation is to single-parameter specifications over a fixed signature. To add parameterisation, the syntax of specification expressions is extended:

$$\begin{array}{rcl} F & ::= & \lambda X{:}\,\mathit{Spec}(\Sigma).\,SP \\ SP & ::= & \ldots \ \mid \ X \ \mid \ F\,SP \end{array}$$

The expression $\lambda X{:}\,\mathit{Spec}(\Sigma).\,SP$ is a *parameterised specification* which can be applied to a specification expression to yield a new specification. The identifier X can be used as a Σ-specification inside the body specification SP.

1.3.1 Type-checking parameterised specifications

To type-check a specification expression, we now use a *context* which can contain an assumption about the type of the variable X. A context Γ is either empty or is a declaration $X : S(\Sigma)$ for some Σ:

$$\Gamma \ ::= \ \langle\rangle \ \mid \ X : S(\Sigma)$$

For ASL, instead of using a context we might simply attach a tag to the variable, writing $X^{S(\Sigma)}$, for example. But when we move to ASL+, an explicit context is more convenient because it can contain interdependent variables that are declared in sequence.

Three new type-checking rules are shown in Figure 1.2. The first types a variable in a context which declares it, and the second types a parameterised specification, with a type of the form $S(\Sigma) \Rightarrow S(\Sigma')$. This says that it is a

$$\frac{}{X : \mathsf{S}(\Sigma) \blacktriangleright X \Rightarrow \mathsf{S}(\Sigma)}$$

$$\frac{X : \mathsf{S}(\Sigma) \blacktriangleright SP \Rightarrow \mathsf{S}(\Sigma')}{\blacktriangleright \lambda X{:}\, \mathit{Spec}(\Sigma).\, SP \Rightarrow \mathsf{S}(\Sigma) \Rightarrow \mathsf{S}(\Sigma')}$$

$$\frac{\blacktriangleright F \Rightarrow \mathsf{S}(\Sigma) \Rightarrow \mathsf{S}(\Sigma') \qquad \blacktriangleright SP \Rightarrow \mathsf{S}(\Sigma)}{\blacktriangleright F\, SP \Rightarrow \mathsf{S}(\Sigma')}$$

Figure 1.2: Type-checking ASL parameterisation

mapping from Σ-specifications to Σ'-specifications. The third rule, for applying a parameterised specification with type $\mathsf{S}(\Sigma) \Rightarrow \mathsf{S}(\Sigma')$, checks that the argument specification has the signature Σ.

We still need to use the rules of Figure 1.1 to type the ASL operators, but they must be modified to contain a context Γ. For example, the rule for **impose** becomes:

$$\frac{\Gamma \blacktriangleright SP \Rightarrow \Sigma \qquad \Phi \subseteq \mathbf{Sen}(\Sigma)}{\Gamma \blacktriangleright \mathbf{impose}\ \Phi\ \mathbf{on}\ SP \Rightarrow \Sigma}$$

Although the context plays no part in any of the rules of Figure 1.1, it must be added to each typing premise and conclusion to record any assumption about the variable X.

The rules of Figure 1.2 bear an obvious resemblance to the type-checking rules of the simply-typed λ-calculus, $\lambda^{\rightarrow}$ [Hindley and Seldin, 1987, Barendregt, 1992]. But they are more restrictive, because $\lambda^{\rightarrow}$ allows contexts of more than one variable, and the variables are allowed to have function types. This corresponds to *higher-order parameterisation*, which is introduced in ASL+.

1.3.2 Semantics of parameterised specifications

There are a couple of choices available for giving a semantics to parameterised specifications. We may choose to interpret $\lambda X{:}\, \mathit{Spec}(\Sigma).\, SP$ purely as a syntactic notation for *macro definition*, and define application by:

$$(\lambda X{:}\, \mathit{Spec}(\Sigma).\, SP)\ SP_{arg} =_{\text{def}} SP[SP_{arg}/X]$$

where $SP[SP_{arg}/X]$ is the substitution of SP_{arg} for X in SP.

Oriented from left to right, the equation above is the β-reduction relation of the λ-calculus. Examining the rules in Figures 1.1 and 1.2, it is easy to show the important *subject reduction* property of the type system. If

$$\blacktriangleright\ \lambda X{:}\,\mathit{Spec}(\Sigma).\,SP \implies S(\Sigma) \Rightarrow S(\Sigma')$$

and

$$\blacktriangleright\ SP_{arg} \implies S(\Sigma)$$

then we also have

$$\blacktriangleright\ SP[SP_{arg}/X] \implies S(\Sigma').$$

This says that β-reduction is sound for typing.

Moreover, we can show that by repeatedly β-reducing, any specification expression can be reduced to a unique *normal form* which is a term without any applications. This means that β-reduction provides an *operational semantics* for specification expressions.

Defining application as macro-expansion, the term $\lambda X{:}\,\mathit{Spec}(\Sigma).\,SP$ itself does not have a model-theoretic denotation. An alternative semantics of parameterisation defines $[\![\lambda X{:}\,\mathit{Spec}(\Sigma).\,SP]\!]$ as a function on model classes, namely the function f such that

$$f(k) = [\![SP]\!]_{[X \mapsto k]}$$

for all $k \subseteq \mathbf{Mod}(\Sigma)$. Here $[\![SP]\!]_{[X \mapsto k]}$ is the interpretation of SP in the environment which assigns X to the model class k; this is similar to the environment model semantics for the λ-calculus. Now we have a *denotational semantics* for *all* specification expressions.

An important property of an environment model semantics is the relation between syntactic substitution and updating the environment, namely that:

$$[\![M[N/x]]\!]_\eta = [\![M]\!]_{\eta[x \mapsto [\![N]\!]]}$$

Typically, if this property holds we can show soundness of β-reduction, which proves that the operational semantics is sound for the denotational semantics.

Until now, most studies of parameterisation in specification languages have had a bias towards either an operational semantics or a denotational semantics. The range goes from the macro-expansion idea above; through so-called *presentation-level semantics* in which the meaning of parameterisation is defined as a function on the *syntax* for specification expressions; to wholly denotational approaches like *pushout parameterisation*, where a parameterised specification itself is a semantic entity, and application is defined using a diagram in the category of specifications.

One of my aims in pursuing the λ-calculus approach to parameterisation is to provide a framework which provides both an operational and a denotational interpretation of the whole language, in the sense described above. Each language I shall study has a context-sensitive equational theory which includes β-equality, and which must be proved sound for the semantics. Operational semantics is not be considered explicitly apart from considering the important properties of *strong normalization* and *subject reduction.*

1.4 ASL+

ASL+ was introduced by Sannella et al. [1992] as an extended version of ASL, although the choice of specification building operators is orthogonal to the main design of ASL+. As a brief introduction to the language, I shall review the syntax of ASL+ given by Sannella et al. [1992].[3] The use of ASL+ is demonstrated by example in the next chapter, and two new versions of ASL+ are presented in detail later in the thesis, in Chapters 5 and 7.

ASL+ departs radically from ASL by adding programs to the syntax, along with constructors for λ-abstraction, Π-abstraction, and the *Spec* constructor which builds power sets.

The expressions in ASL+ are called *pre-terms*, and are ranged over by several meta-variables. We use $M, N, \ldots$ for arbitrary pre-terms and SP for pre-terms which are intended to denote specifications, although there is no difference formally. Other variables like A, B, F, G, H are also used. Pre-terms are formed by a context-free grammar:

$$\begin{array}{rcl} M, SP & ::= & X \mid \{M\} \mid \lambda X{:}SP.\, M \mid M\, M \\ & \mid & \Pi X{:}SP.\, SP \mid \mathit{Spec}(SP) \\ & \mid & P \\ & \mid & \Sigma \mid \textbf{impose } \Phi \textbf{ on } SP \mid \ldots \end{array}$$

The last line is continued with the syntax of ASL from Section 1.2. The new meta-variable P ranges over programs written in some unspecified institution-dependent syntax. Borrowing terminology from SML, I sometimes call programs P and ASL specifications *core-level* programs and specifications, to distinguish from the *module-level* part of the language provided by the ASL+ λ-calculus. (The separation between the two levels is discussed more in Section 6.9.3.)

[3]With one minor difference: the union operator here is left at the level of ASL terms, as it was in Sannella et al. [1990].

ASL had separate syntactic categories for parameterised specifications and ordinary specifications, but ASL+ has a single syntactic category of preterms, which may be programs, specifications, or parameterised objects; context-sensitive type-checking rules are used to determine what kind of object something is. (This mixing of syntax into a single category of terms and the use of several meta-variables is typical in complex type theories.)

Here is a description of the operators of ASL+.

X is a variable; it only makes sense with respect to a context, because variables can range over programs, specifications, parameterised programs, etc.

$\{M\}$ is the tight *singleton specification* which is satisfied uniquely by M. Singletons allow a program to turned into a specification, providing the crucial (and only) connection between the two.

More details of singleton types are given in Chapter 3.

$\lambda X{:}SP.\,M$ is a parameterised object, which can be applied to terms satisfying the specification SP. For example, the term

$$\lambda X{:}\,\mathsf{Spec}\,(SP).\,SP'$$

is a parameterised specification which can be applied to refinements of SP. In ASL+ parameters can have *semantic requirements*, unlike ASL where parameters always have the form $\mathsf{Spec}(\Sigma)$ for some signature Σ, amounting only to type-checking requirements. Another example of parameterisation is the term

$$\lambda X{:}SP.\,P$$

which is a parameterised program that can be applied to programs which satisfy SP. Semantically it denotes a function on algebras.

$M\ N$ is the application of a parameterised object M to an object N. For this to be meaningful, we must check that M indeed denotes a function, and that N denotes a value in its domain. This is the crux of the type-checking and satisfaction systems we shall study.

$\Pi X{:}SP.\,SP'$ is the specification of a parameterised object. Semantically, it denotes the collection of functions f which map elements m of $[\![SP]\!]$ to elements $f(m)$ of $[\![SP']\!]_{[X\mapsto m]}$. The dependency in the Π-term means we can specify functions which depend on their arguments.

$\mathsf{Spec}(SP)$ is the specification of refinements of SP. If SP denotes a set, $\mathsf{Spec}(SP)$ denotes the power set of SP. This constructor was devised for similar purposes as Cardelli's idea of a *power type* [Cardelli, 1988], written as $\mathsf{Power}(A)$ — it allows a single form of λ-abstraction to express both kinds of parameterisation shown above.

More details of power types are given in Chapter 4.

Several abbreviations will be used for terms. The term $\lambda X{:}A\ Y{:}B.C$ abbreviates $\lambda X{:}A.\lambda Y{:}B.C$ and $\lambda X, Y{:}A.C$ abbreviates $\lambda X{:}A\ Y{:}A.C$. Similarly for Π-terms. Sometimes I write applications $F\,M$ using parentheses, as $F(M)$, and the curried application $F\,M\,N$ is sometimes written as $F(M, N)$.

The type-checking system, semantics, and satisfaction system for ASL+ are developed in the rest of the thesis.

1.5 *Pre-requisites and Conventions*

I have tried to keep the pre-requisites to a minimum, and to make the material accessible to readers from either the algebraic specification camp or the type theory camp. Standard references for each field may be helpful, for example [Wirsing, 1990] and Barendregt [1992].

Here are some notational conventions used in the thesis.

Set notation. If S is a set, $Pow(S)$ denotes the power set of S and $Fin(S)$ denotes the set of finite subsets of S.

The set of total functions between two sets S and T is written as $S \to T$ and the set of partial functions between the two sets as $S \rightharpoonup T$. Often I write $f : S \to T$ instead of $f \in S \to T$. If $f \in S \rightharpoonup T$, then the domain of S is written $Dom(S)$. The "updated" function $f[s \mapsto t]$ is defined by

$$f[s \mapsto t](x) = \begin{cases} t & \text{if } x = s, \\ f(x) & \text{otherwise.} \end{cases}$$

Grammars. When specifying context-free grammars I sometimes use meta-variables suggestively, rather than the names of the sets they range over. For example,

$$\begin{array}{rcccl} n & ::= & 0 & \mid & succ(n) \\ i & ::= & n & \mid & -n \end{array}$$

rather than

$$\begin{array}{lcl} nat & ::= & 0 \mid succ(nat) \\ int & ::= & nat \mid -nat \end{array}$$

Preceding sections used this convention already; it avoids the need to introduce names for syntactic categories.

1.6 Outline of the Thesis

Each chapter begins with an overview and table of contents and ends with a summary and a comparison to the relevant related work. Here is an outline.

Chapter 2 motivates the design of ASL+ by sketching an example of modular formal program development. The example demonstrates specifications parameterised on specifications and programs. It also demonstrates parameterised programs and specifications of parameterised programs. Some of the features of ASL+, and the concerns with their formalization, are introduced. In part, the quest of the thesis is to make the underpinnings of this example completely formal.

Chapter 3 is the first of two fundamental studies of new typed λ-calculi which are sub-systems of ASL+. This chapter introduces a type-system called $\lambda_{\leq\{\}}$ which has subtyping and *singleton* types. Some basic concepts of subtyping are introduced, and good properties of the calculus are established, including *subject reduction* and *minimal typing.* The calculus is given a PER model semantics, which is the first PER semantics of a system with dependent types and subtyping. The typing rules are proved sound in the model.

Chapter 4 is the second fundamental study, and introduces a type-system called λ_{Power}. This is a predicative fragment of Cardelli's 1988 power type system, containing only power types and the Π-type dependent product constructor. This system is shown to have some expected basic properties and is given a companion system for *rough type-checking,* based on non-dependent types. Rough typing enjoys the subject reduction property and uniqueness of types. The full system and rough typing have the *agreement* property mentioned before; this is used to establish that the full system is strongly normalizing. Subject reduction for the full system is not proved, however. Rough types are used to structure a model definition for λ_{Power}, given as a novel form of applicative structure. This improves on the model in Chapter 3 which despite being a small extension of the simply-typed calculus, is based on a λ-model, following earlier work on subtyping calculi. (In Chapter 5, singleton types are added to the applicative structure model definition.) Several examples of λ_{Power} models are described.

Chapter 3 and Chapter 4 are self-contained, and can each be read independently of the rest of the thesis.

Chapter 5 revisits some of the basic ideas behind ASL+, showing how the abstract language would be used for a real language, before reformulating it to solve problems with the original definition given by Sannella et al. [1992]. Things begin from $\lambda_{\mathsf{ASL+}}$, a combination of the syntax of λ_{Power} and $\lambda_{\leq\{\}}$ with unspecified sets of program and specification building operators. The rough typing system of $\lambda_{\mathsf{ASL+}}$ is defined first, and the model definition of λ_{Power} is extended to a definition of model for $\lambda_{\mathsf{ASL+}}$. The satisfaction system for $\lambda_{\mathsf{ASL+}}$ is based on the type systems of λ_{Power} and $\lambda_{\leq\{\}}$, augmented with a consequence relation for proving things about the program and specification building operators. ASL+ itself is defined by a particular choice of the program and specification building operators and a particular consequence relation. The specification building operators are chosen to be the institution-independent operators of ASL shown in Section 1.2.

The second part of Chapter 5 highlights problems with this abstract treatment of ASL+, which make it difficult to use rough typing to explain the composition of modular programs. (It is because rough typing is based on simple types, so cannot explain the propagation of type identities.) The example of Chapter 2 does not really work yet, and more effort is needed to formalize it.

Chapter 6 begins an effort towards formalizing ASL+ examples directly, following the maxim that to understand the general we must first understand the specific. This chapter defines a new institution which captures the programming language and logic used in Chapter 2. The institution is $\mathcal{FPC}$, based on the classical domain theoretic semantics of FPC, a tiny but powerful functional programming language [Gunter, 1992, Plotkin, 1985] which has partial recursive functions and recursive types. The logic of $\mathcal{FPC}$ is Higher Order Logic, extended with LCF-style constructs for reasoning about the types of FPC. Concrete syntax and type-checking rules are given for signatures, signature morphisms, and algebras in $\mathcal{FPC}$. An important novelty is that signatures in $\mathcal{FPC}$ can have type equations which are used in type-checking and restrict the range of models.

Chapter 7 builds a concrete version of ASL+ on top of $\mathcal{FPC}$, called ASL+$_{\mathsf{FPC}}$. The rough type system of ASL+$_{\mathsf{FPC}}$ is more expressive than the one studied in Chapter 5, because it incorporates a form of dependent types. Because signatures in $\mathcal{FPC}$ have type equations, we can now directly explain the examples in Chapter 2. A new semantics for this version of ASL+ is given, based on the enhanced rough typing system. Some ideas for a satisfaction system for ASL+$_{\text{FPC}}$ are sketched.

Chapter 8 reviews what has been achieved and sets forth some ideas for future work.

Appendix A provides some details about the formulation of the language FPC, extending the coverage in Chapter 6, and it gives some details of a provisional satisfaction system for ASL+$_{\text{FPC}}$, following the ideas in Chapter 7.

2 An Example in ASL+

This chapter sketches an example to motivate and demonstrate the mechanisms of ASL+ for expressing and developing the large-scale structure of software systems.

The example is a keyword-in-context index, which is traditional for demonstrating modular programming since Parnas's seminal paper [Parnas, 1972]. It consists of only a small number of modules, but is enough to illustrate some salient features of ASL+.

The example is pitched at an intuitive and slightly informal level. In part, the quest of the following chapters is to demonstrate that ASL+ and this example can be given a fully formal underpinning.

2.1 Requirements Specification

THIS EXAMPLE DEMONSTRATES the use of ASL+ in a small development of a modular program from a modular specification. The program is written in a functional programming language, and the specification uses a higher-order logic for writing axioms. Similar but smaller examples were undertaken by Sannella et al. [1990] and Sokołowski [1989].

This first section describes writing the requirements specification from an informal description of the problem: this specification is to be delivered by a customer to a software house for formal development of an implementation. That second phase is described in Section 2.2. The intention is to demonstrate the use of the language, rather than show a perfect example of program development. Because of this, some bad design choices are taken deliberately.

2.1.1 Problem description

The problem is to generate a keyword-in-context index from a collection of titles. A title is a sequence of words; some words are considered significant and called *keywords*. The index should consist of titles with a single emphasised keyword. It is sorted by keyword, and where a keyword *k* occurs in several titles, the order is determined by forming the "circular shift" of the keywords in each title, such that *k* is at the front. The order of titles under *k* is then given by the lexicographic ordering (extending the keyword ordering) of their circular shifts.

For example, given as input the titles of these books:

> Northcott, Jim, *Chips and jobs : acceptance of new technology at work.* London Policy Studies Institute 1985.
>
> Schroeder, Dirk, *Computer software protection and semiconductor chips.* London Butterworths 1990.
>
> Walton, John K., *Fish and chips and the British working class, 1870-1940* Leicester University Press, 1992.

the output would start like this:

> Chips and jobs : **acceptance** of new technology at work
> Fish and **chips** and the British working class, 1870-1940
> Computer software protection and semiconductor **chips**
> **Chips** and jobs : acceptance of new technology at work
> Fish and chips and the British working **class**, 1870-1940
> ⋮

assuming that "at," "and," "of," and "the" are non-keywords.

2.1.2 Formalising the description

The first decision is what sort of entity we want delivered as an implementation. We may already have a datatype for titles and keywords in mind, or even a specific set of titles that we want to generate an index for. But if we ask the software house for an implementation which is *generic* as far as possible, we may generate other indexes later, or re-use the indexing function in another setting.

We shall build a specification `KWICFUN` which specifies a program parameterised on datatypes for *words*, *keywords*, *emphasised words*, *titles*, and *emphasised titles.* Given implementations of these types, the program should return a function to generate a keyword-in-context index from a set of titles.

2.1.3 Library specifications

Before beginning the design of the requirements specification, we mention a couple of specifications which might be part of a standard library.

As well as assuming a library of specifications, we assume that the programming language has some *pervasive* types, which are standard implementations of certain library specifications. For example, it would be unusual or even impossible to re-implement the standard type *bool* of booleans in a programming language. We shall assume that these pervasive types are declared in a initial context and have standard properties available, in other words, they are assumed to satisfy the standard library specifications. Of course, it's absolutely crucial to the soundness of the development that the properties of such pervasive types really hold in the programming language used.

This treatment of pervasive types seems more realistic than in specification languages which must import the specification BOOLEAN of booleans into every specification. Besides avoiding such verbosity, we assume that the same implementation of *bool* is used everywhere. It means that specifications and algebras can be *open* because they refer to sort and operation names which are not declared locally.

As well as *bool*, the pervasive types will include: strings *string*; product types $\alpha \times \beta$ for types α, β with projections *fst*, *snd*; and the type $option_\alpha$ for any type α, which has constructors *none* and *some*(e).

To build parameterised datatypes, we often use the trivial specification ELT of algebras with a single carrier `elt`:

```
ELT =def spec
           type elt
         end
```

The parameterised specification ORD maps refinements of ELT to specifications in which `elt` is a total order:

```
ORD =def λ ELT' ≤ ELT .
              enrich ELT'
                with spec
                       val _ ≤ _: elt × elt → bool
                       axioms
                         x ≤ x
                         x ≤ y ∧ y ≤ x ⇒ x = y
                         x ≤ y ∧ y ≤ z ⇒ x ≤ z
                         x ≤ y ∨ y ≤ x
                     end
```

As usual, axioms are assumed to be implicitly universally quantified over their free variables. In ASL+, the notation $\lambda X \le SP.\,SP'$ is shorthand for $\lambda X\colon \mathit{Spec}\,(SP).\,SP'$.

Specifications of lists come in two flavours: a specification of a parameterised program, LISTFUN and a parameterised specification, LIST.

```
LISTFUN =def
  Π Elt : ELT .
      spec
        type elt = Elt.elt
        type list
        val nil : list
        val cons : elt × list → list
        val null : list → bool
        val hd : list → option_elt
        val tl : list → option_list
        axioms
          null(nil) = true
          null(cons(e, l)) = false
          hd(cons(e, l)) = some(e)
          tl(cons(e, l)) = some(l)
          hd(nil) = none
          tl(nil) = none
      end

LIST =def
  λ ELT' ≤ ELT .
      enrich ELT'
        with spec
              type list
              val nil : list
              val cons : elt × list → list
              val null : list → bool
              val hd : list → option_elt
              val tl : list → option_list
              axioms
                as above
            end
```

LISTFUN specifies programs which map an argument implementation of ELT into an implementation of lists of elements of the argument type elt, whereas LIST maps refinements ELT' of ELT into specifications of lists over the elt sort. We use LISTFUN when *specifying the design structure of the implementation*, and LIST for *structuring the requirements specification.* The denotations of LISTFUN and LIST are related by a Galois connection which is the main topic studied by Sannella et al. [1992].

We shall use lists throughout the development, although in several places, sets would be a better choice.

2.1.4 Words, keywords, and emphasis

The first datatype we specify is for *words*. The simple requirement is that words are totally ordered, which we specify by renaming an application of the library specification ORD. Additionally, we give a constructor for building words so that we can specify some non-keywords later.

```
WORD =def enrich translate ORD(ELT) by [elt ↦ word]
              with spec
                     val make_word : string → word
                 end
```

The next datatype is for *keywords*. A keyword type is not a fresh datatype; rather we wish to capture the idea that it is an extension of a given word datatype. This is λ-abstraction rather than Π-abstraction, since we take a *particular* word type to build the specification, rather than specifying a parameterised program to build the datatype for *any* word type. The situation is similar to the pervasive types like *bool*: we want to base the specification on a particular implementation of WORD. In this case the implementation is made into a parameter of the specification, rather than being fixed everywhere.

Given an implementation Word : WORD, we specify a predicate is_keyword defined on the carrier of Word:

```
KEYWORD =def
   λ Word : WORD .
      spec
         type word = Word.word
         val is_keyword : word → bool
         axioms
            is_keyword(Word.make_word("at")) = false
            is_keyword(Word.make_word("and")) = false
            is_keyword(Word.make_word("of")) = false
            is_keyword(Word.make_word("the")) = false
      end
```

The type equation word = Word.word specifies that the specification should require the same carrier for word as was passed in the parameter.

The type of *emphasised words* is defined to be the product of a given type of words and a type of emphasis decorations. Emphasis decorations include a value for indicating keywords, keyword, and one for unadorned words, plain.

```
EMPHWORD =def
  λWord : WORD .
      spec
         type emphasis
         val keyword : emphasis
         val plain : emphasis
         type word = Word.word × emphasis
      end
```

The type equation word = Word.word × emphasis specifies the implementation of word concretely, rather than axiomatically. This fixes the implementation of word in any application of EMPHWORD.

2.1.5 Dot notation and type equations

The "dot notation" used above in the LISTFUN and KEYWORD specifications, and the type equations like word = Word.word used in KEYWORD, are not primitive parts of ASL+ as it was originally conceived. They can be removed by translation into a form which uses the *singleton* constructor:

```
KEYWORD =def
  λWord : WORD .
      derive from
         impose
            { is_keyword(Word.make_word("at")) = false · · · }
         on
               translate {Word} by σ_WORD
         by σ_KEYWORD
```

where $\sigma_{WORD} : \Sigma_{WORD} \to \mathtt{Word.}\Sigma_{WORD} \cup \{\,\mathtt{is\_keyword}\,\}$ is given by

$$\sigma_{WORD} =_{\text{def}} [\,\mathtt{word} \mapsto \mathtt{Word.word}, \mathtt{make\_word} \mapsto \mathtt{Word.make\_word}\,]$$

and $\sigma_{KEYWORD} : \Sigma_{KEYWORD} \to \mathtt{Word.}\Sigma_{WORD} \cup \{\,\mathtt{is\_keyword}\,\}$ is given by

$$\sigma_{KEYWORD} =_{\text{def}} [\,\mathtt{word} \mapsto \mathtt{Word.word}, \mathtt{is\_keyword} \mapsto \mathtt{is\_keyword}\,]$$

Here Σ_{WORD} is the signature of WORD, $\Sigma_{KEYWORD}$ is the signature of the body of KEYWORD, and Word.Σ_{WORD} stands for the signature Σ_{WORD} renamed by prefixing the components with "Word.".

This translation explicitly ensures that the body of the parameterised specification is specified as an extension of the argument algebra Word, which uniquely satisfies {Word}. A similar way of handling dot notation was suggested in the appendix of Sannella et al. [1990].

This translation testifies to the power of the kernel language, but it isn't clear that it is advisable. The type-checker must pay careful attention to the renaming of symbols in this way, and the translation has the flavour of an ad-hoc "hack" needed to manage the current context of declarations which arise from λ- or Π-abstraction. Furthermore, for more complicated examples like EMPHWORD, this renaming approach is *not* sufficient, because we wish to express equality of part of a type expression, rather than just the identity of names. In that case it seems that we need a richer notion of signature morphism, or a special kind of axiom in the logic which can express equality of types.

Chapter 7 gives the scheme a more satisfactory basis, so that an algebraic signature is generated according to the current context, which contains the symbols renamed appropriately. The semantics is made aware of the notion of context, and signatures are extended to include type equations like those above.

2.1.6 Titles and emphasised titles

Let's return to the specification. Next we define a datatype for *titles*, which is parameterised on an implementation of WORD and an implementation of lists of words. Titles have a destructor for giving the list of words in a title, and a constructor for creating a title from a list of words.[1]

```
TITLE =def
  λ Word : WORD
    WordList : LIST (spec type elt = Word.word end) .
      spec
        type title
        val words_of_title : title → WordList.list
        val make_title : WordList.list → title
        axiom words_of_title(make_title(s)) = s
      end
```

The term

```
LIST (spec type elt = Word.word end)
```

constructs a specification of lists over the type Word.word.

Emphasised titles are like titles, except they contain emphasised words instead of words. It would be nice to re-use the TITLE specification to specify emphasised titles, by writing:

[1]Recall that $\lambda X{:}A\,Y{:}B.\,C$ is an abbreviation for $\lambda X{:}A.\,\lambda Y{:}B.\,C$.

```
λ Word : WORD
  EmphWord : EMPHWORD(Word)
  EmphWordList : LIST (spec type elt = EmphWord.word end) .
    TITLE (EmphWord, EmphWordList)
```

Unfortunately, this doesn't quite work because the signature of EmphWord is not compatible with that of WORD, since it does not contain a make_word operator. To amend this, we could either add a make_word function to EMPHWORD, or as suggested already, remove make_word from WORD since it is perhaps over-specifying WORD to include it in the first place.

Instead we give a similar specification to TITLE, defining:

```
EMPHTITLE =def
  λ Word : WORD
    EmphWord : EMPHWORD(Word)
    EmphWordList : LIST (spec type elt = EmphWord.word end) .
      spec
        type title
        val words_of_title : title → EmphWordList.list
        val make_title : EmphWordList.list → title
        axiom words_of_title(make_title(s)) = s
      end
```

A different specification for EMPHTITLE would be to *fix* the implementation type based on some implementation of titles, in a similar way to the specification of words and emphasised words. For example,

```
λ Word : WORD
  WordList : LIST (spec type elt = Word.word end)
  Title : TITLE(Word, WordList)
  EmphWord : EMPHWORD(Word)
  EmphList : LIST (spec type elt = EmphWord.emphasis end) .
    spec
      type title = Title.title × EmphList.list
    end
```

This specifies a concrete implementation of emphasised titles.

2.1.7 Target specification

We can now give the main specification, which has three axioms:

```
KWICFUN =def
  Π List : LISTFUN
```

```
Word : WORD
Keyword : KEYWORD (Word)
EmphWord : EMPHWORD (Word)
Title : TITLE (Word, List (alg type elt = Word.word end))
EmphTitle :
  EMPHTITLE (Word, EmphWord,
             List (alg type elt = EmphWord.word end))
TitleList : LIST(spec type elt = Title.title end)
EmphTitleList : LIST (spec type elt = EmphTitle.title end) .
  spec
    val make_kwic : TitleList.list → EmphTitleList.list
    axioms
      make_kwic(ts) consists of titles in ts with exactly one key-
      word emphasised.

         $e \in_{\text{list}}$ make_kwic($ts$) $\Longrightarrow$
           single_keyword_emphasised($e$)
             $\bigwedge$ title_of_emphtitle($e$) $\in_{\text{list}}$ $ts$

      Every title appears in the output exactly once for each key-
      word occurrence, with that occurrence emphasised.

         $t \in_{\text{list}} ts \Longrightarrow$
           $\forall k$.
             Keyword.is_keyword(Title.words_of_title($t$)$_k$)
             $\Longrightarrow \exists! e.\ e \in_{\text{list}}$ make_kwic($ts$)
             $\bigwedge$ title_of_emphtitle($e$) = $t$
             $\bigwedge$ fst(EmphTitle.words_of_title($e$)$_k$)
                = EmphWord.keyword

      make_kwic(ts) is sorted lexicographically on the circular
      shift of the keywords of each title needed to bring the single
      emphasised word to the front.

         ordered_by(Word.$\leq_{\text{lex}}$ ∘ title_shift,
                    make_kwic($ts$))
    end
```

We assume that the logic available is powerful enough to *define* functions recursively similarly to the way they might be implemented in the destination programming language. The axioms above are given in terms of a number of such auxiliary functions; in a weaker institution these might be specified as hidden functions, but here we assume that local definitions can be expressed in the logic, or simply that the definitions are abbreviations in the meta-language.

The following definitions are used in the axioms:

1. Operations on lists and orderings:

$$
\begin{array}{lll}
l_n & =_{\text{def}} & \mathtt{nth\_element}(l, n) \\
a < b & =_{\text{def}} & a \leq b \ \bigwedge \ \neg \ (b \leq a) \\
l \leq_{\text{lex}} m & =_{\text{def}} & (\exists r. l@r = m) \ \bigvee \\
 & & \quad (\exists t, u, v, w. t < u \\
 & & \quad \bigwedge \ l = s@[t]@v \ \bigwedge \ m = s@[u]@w) \\
a \ (\leq \circ f) \ b & =_{\text{def}} & (fa) \leq (fb) \\
\mathtt{ordered\_by}(\leq, l) & =_{\text{def}} & \forall n. 0 < n < \mathtt{length}(l) \Longrightarrow l_n \leq l_{n+1}
\end{array}
$$

2. Specific operations and predicates:

$$
\begin{array}{l}
\mathtt{title\_of\_emphtitle}(e) =_{\text{def}} \\
\quad \mathtt{Title.make\_title(words\_of\_emphtitle}(e)) \\
\mathtt{emphasis\_of\_emphtitle}(e) =_{\text{def}} \\
\quad \mathtt{map}\ \mathit{fst}\ (\mathtt{EmphTitle.words\_of\_title}(e)) \\
\mathtt{words\_of\_emphtitle}(e) =_{\text{def}} \\
\quad \mathtt{map}\ \mathit{snd}\ (\mathtt{EmphTitle.words\_of\_title}(e)) \\
\mathtt{single\_keyword\_emphasised}(e) =_{\text{def}} \\
\quad \exists ! w.\ (\mathtt{emphasis\_of\_emphtitle}(e))_w = \mathtt{EmphWord.keyword} \\
\qquad \bigwedge \ \mathtt{Keyword.is\_keyword}((\mathtt{words\_of\_emphtitle}(e))_w) \\
\mathtt{title\_shift}(e) =_{\text{def}} \\
\quad \epsilon l.\ l = [\mathtt{emph\_word}(e)]@m@n \\
\qquad \bigwedge \ \mathtt{keywords\_of\_emphtitle}(e) = n@[\mathtt{emph\_word}(e)]@m \\
\mathtt{keywords\_of\_emphtitle}(e) = \\
\quad \mathtt{filter}\ \mathtt{Keyword.is\_keyword}\ (\mathtt{words\_of\_emphtitle}(e)) \\
\mathtt{emph\_word}(e) =_{\text{def}} \\
\quad \mathbf{snd}(\mathtt{nth\_element}(\epsilon w. (\mathtt{emphasis\_of\_emphtitle}(e))_w = \mathtt{keyword}, \\
\qquad\qquad\qquad \mathtt{EmphTitle.words\_of\_title}(e)))
\end{array}
$$

The symbol $\in_{\text{list}}$ denotes a membership function on lists; @ is the append operator on lists; ϵ is a logical choice operator such that $P(\epsilon x.P(x))$ provided $\exists x.P(x)$. These definitions are quite informal in that we heavily overload symbols and have omitted to say where many of them come from. For example, `map` is a function that can be defined for any two implementations `List1` and `List2` of lists:

$$
\begin{array}{l}
\mathbf{val}\ \mathtt{map} : \mathtt{List1.list} \to \mathtt{List2.list} =_{\text{def}} \\
\quad \lambda f : \mathtt{List1.elt} \to \mathtt{List2.elt}.
\end{array}
$$

```
μmap.λl.if List1.null(l)
        then List2.nil
        else List2.cons(f(List1.hd(l)), map(List1.tl(l)))
```

The append operator @, the function `nth_element`, and the function `filter` would be defined similarly.

There is a certain amount of (automatic) type-checking that may be done on the specification `KWICFUN` to ensure its well-formedness, which might be the obligation of the customer. On the other hand, well-definedness of the specification, in the sense of consistency, will be demonstrated by the software house's ability to deliver an implementation `Kwic` such that `Kwic : KWICFUN`.

2.1.8 Over-specification?

From the viewpoint of the final `KWICFUN` specification, we have probably *over-specified* `WORD` and `KEYWORD` by including `make_word` and the related axioms.

The task of the software house is to provide a parameterised program which implements an index given any implementations of `WORD` and `KEYWORD`. Restrictions on which words are keywords, for example, may be requirements the customer wishes to impose on the datatypes later, but it is perhaps misguided to deliver these as part of the `KWICFUN` specification itself.

The "accidentally delivered" information probably isn't necessary for a good implementation, and the customer may be surprised if an implementation made use of it. There might be a problem if `KWICFUN` was needed later for indexes where the word "the" was to be considered a keyword after all, for example.

Fortunately, the situation can be rectified at a later point using a rule for "contravariant refinement" to adjust the requirement specification; see Section 2.2.1.

2.1.9 Discussion

In this section we discuss a couple of the design issues behind the `KWICFUN` specification.

Forms of parameterisation. `KEYWORD` and some of the following specifications show a perhaps unusual form of parameterisation; a specification parameterised upon an algebra. An alternative might be to define `KEYWORD` instead as a parameterised specification, with heading λW ≤ WORD, and then apply keyword to a singleton argument when necessary, KEYWORD({Word}).

But we don't require the extra flexibility of a parameterised specification in this example. Even if we were to need keywords over a refinement WORD' ≤ WORD, every Word : WORD' is a permissible argument to KEYWORD as it stands, by the *subsumption* principle of subtyping systems: $M : SP$ and $SP \le SP'$ implies $M : SP'$:

$$\frac{M : SP \qquad SP \le SP'}{M : SP'}$$

The language SPECTRAL also features specifications parameterised on algebras and claims that user-defined specifications parameterised on specifications are not needed [Krieg-Brückner and Sannella, 1991]. (The examples here seem to back that opinion, more or less, but hardly constitute sufficient evidence.)

Sharing implementations. I suggested above that it is natural to regard keywords and the other datatypes as defined with respect to a given implementation of WORD. If this wasn't observed, we might instead write:

```
KEYWORD' =def enrich WORD
                with spec
                       val is_keyword : word → bool
                       axiom is_keyword(make_word("a")) = false  ...
                     end
```

But then we would run into problems when writing KWICFUN. Because KWICFUN has several parameters which are related, rather than a single monolithic parameter, we need some way to specify that parts of the parameters *share* the same implementation, so that certain types are equal. In the second axiom of KWICFUN, for example, the term Keyword.is_keyword is applied to a word from Title. For this to type-check, Keyword.word and Title.word must be equal types.

Here the parameterisation of KEYWORD, together with the type equations, enforces the sharing needed; this is *sharing by parameterisation*, like in Pebble [Lampson and Burstall, 1988].

By contrast, the example in Sannella et al. [1990] expressed sharing via extensional equality on sorts in special axioms; a similar approach is via the *sharing constraints* of Standard ML, which are specially restricted equations between components. Sharing constraints would be needed if we used the specification KEYWORD' instead of KEYWORD. Compared with sharing by parameterisation, sharing constraints have a *post-hoc* flavour, because one does not need to anticipate which parts of a module to parameterise over in advance. Of course, this can cause unexpected problems when combining modules at a late stage which do not satisfy their sharing constraints; the benefit of extra parameterisation is that it allows more flexibility in the development order.

Perhaps sharing constraints are more suitable for "fine-grain" sharing of components of modules, whilst extra parameterisation is more suitable for expressing sharing of modules themselves. In any case, the vital thing is that the type-checker can understand the propagation of type equations, since these are needed for type-checking axioms, and when it comes to implementation, for type-checking function definitions. The issues behind the task of type-checking have implications which reach to the heart of the semantics of the language, and will be explored towards the end of Chapter 5 and in Chapter 7.

2.2 *Formal Development*

Now we assume the role of the supplier whose job it is to implement the specification KWICFUN. We begin the refinement of the specification KWICFUN towards a concrete implementation, which will be provably correct by virtue of having been built from provably correct components via correctness preserving composition operators.

2.2.1 *Delivering a better product*

On receiving the KWICFUN specification, let's suppose the head programmer at the software house looks at the WORD and KEYWORD specifications, and decides that the presence of make_word and the axioms in KEYWORD comprise over-specification. This is a subjective decision a programmer may make when given a formal implementation task. The pruning of suspect information unnecessary for the job in hand will prevent other programmers from seeing and perhaps exploiting it.

The spurious detail is removed by editing the original specifications:

```
WORD₁ =def translate ORD(ELT) by [elt ↦ word]


KEYWORD₁ =def λ Word : WORD₁ .
                  sig
                     val is_keyword : Word.word → bool
                  end
```

```
EMPHWORD₁, TITLE₁, EMPHTITLE₁
    each as before, except parameterised on -_1 variants.
```

The main specification has now becomes more restrictive, because the domain of the function space has become larger:

```
KWICFUN₁ =def
   Π List : LISTFUN
     Word : WORD₁
     Keyword : KEYWORD₁ (Word)
     EmphWord : EMPHWORD₁ (Word)
     Title : TITLE₁ (Word, List (alg type elt = Word.word end) )
     EmphTitle :
       EMPHTITLE₁ (Word, EmphWord,
                   List (alg type elt = EmphWord.word end) )
     ⋮ . (as before)
```

The correctness of this kind of refinement, by removal of detail from a function argument, is captured formally in ASL+ by the contravariance of the domain of the Π-constructor in the subtyping rule:

$$\frac{S' \leq S \qquad \begin{array}{c}[X : S'] \\ \vdots \\ T \leq T'\end{array}}{\Pi X{:}S.\,T \leq \Pi X{:}S'.\,T'}$$

With this rule we can show $\mathsf{KWICFUN}_1 \leq \mathsf{KWICFUN}$ (so $\mathsf{KWICFUN} \rightsquigarrow \mathsf{KWICFUN}_1$), using several stages:

$$\mathsf{WORD} \leq \mathsf{WORD}_1$$

then

$$\begin{array}{c}[\mathsf{Word} : \mathsf{WORD}_1] \\ \vdots \\ \mathsf{KEYWORD}\,(\mathsf{Word}) \leq \mathsf{KEYWORD}_1\,(\mathsf{Word})\end{array}$$

and similarly for the parameters based on EMPHWORD, TITLE, EMPHTITLE and the rest of KWICFUN.

Removal of trivial constructors. An alternative way of expressing the refinement above is via a *constructor implementation* [Sannella and Tarlecki, 1988b]:

$$\mathsf{KWICFUN} \underset{\kappa}{\rightsquigarrow} \mathsf{KWICFUN}_1$$

where

$$\kappa = \lambda f : \mathsf{KWICFUN}_1 . \lambda l\, w\, k\, e\, t\, e'\, t'\, l' \, . \, f(l, w, k, e, t, e', t', l')$$

and κ is proved to map elements of KWICFUN$_1$ into elements of KWICFUN. We would rather elide the use of trivial constructors like κ, however, since $\kappa =_\eta id_{\mathtt{KWICFUN}_1}$.

Refinement of parameterised specifications In the same way that we show the refinement between the two Π-specifications KWICFUN$_1$ and KWICFUN, we might extend the notion of implementation to parameterised terms, for example to say that KEYWORD $\leq$ KEYWORD$_1$.

Given two parameterised terms, $P = \lambda X{:}SP.\, M$ is a refinement of $P' = \lambda X{:}SP'.\, M'$ when SP refines to SP' (so every argument valid for P' is also valid for P) and for every N in SP', P refines P' "pointwise", $P(N) \leq P'(N)$.

$$\frac{SP' \leq SP \qquad \begin{array}{c}[X : SP']\\ \vdots \\ M \leq M'\end{array}}{\lambda X{:}SP.\, M \leq \lambda X{:}SP'.\, M'}$$

Refinement of parameterised terms like this is similar to certain higher-order subtyping systems where subtyping has been extended to type-operators [Cardelli, 1990, Pierce and Turner, 1994, Steffen and Pierce, 1994]. The idea of weakening the argument specification SP' also occurred to Sannella and Wirsing [1982], who describe a similar notion of refinement of parameterised specifications in CLEAR.

This kind of refinement can already be expressed in the framework, via power types. Instead of asking for a refinement of $P' \equiv \lambda X{:}SP'.\, M'$ we can ask for an implementation of the specification $\Pi X{:}SP'.\, \mathsf{Spec}(M')$. Given such an implementation P, then by the application rule for dependent products types,

$$\frac{P : \Pi X{:}SP'.\, \mathsf{Spec}(M') \qquad N : SP'}{P(N) : \mathsf{Spec}(M'[N/X])}$$

and by β-conversion, $P'(N) = \mathsf{Spec}(M'[N/X])$, so we have $P(N) \leq P'(N)$ for every $N : SP'$ necessarily.

This way of expressing refinement of parameterised objects results in a significant simplification of the type system compared with the higher-order subtyping systems cited above. It is explained in more detail in Section 4.3.2.

2.2.2 Bottom up development: shifting and sorting

The next stage in the development is to ponder what tools could be used to solve the problem. Parnas's original solution was based on two procedures: one to form the circular shifts of the titles and another to sort according to the shifts. We can specify these as two parameterised programs.

The task to implementation the circular shift function is a small self-contained problem:

```
CIRCSHIFTFUN =def
  Π Elt : ELT
    EltList : LIST ({Elt})
    EltListList : LIST (spec type elt = EltList.list end) .
      spec
        val circ_shifts : EltList.list → EltListList.list
        axiom
        circ_shifts(l) consists of all circular shifts of l
          m ∈list circ_shifts(l)
              ⟺ ∃a, b.l = a@b ∧ m = b@a
      end
```

The programmer who tackles this task should return with a program `Circshift` and a proof that:

```
Circshift : CIRCSHIFTFUN
```

Sorting, meanwhile, is specified and formally developed so often that any reputable software house must have many examples of formally developed sorting procedures in its library[2]. We assume that sorting programs are parameterised upon the ordering to be sorted and an implementation of lists, and satisfy a specification of the form:

```
SORTFUN =def
  Π Elt : ORD(ELT)
    EltList : LIST (Elt) .
      spec
        val sort : EltList.list → EltList.list
        axioms
          sort(l) is a sorted copy of l, with respect to Elt.≤
            a ∈list l ⟺ a ∈list sort(l)
            repetitions(a, l) = repetitions(a, sort(l))
            ordered_by(Elt.≤, sort(l))
      end
```

Again, the functions `repetitions` and `ordered_by` are assumed to be defined concretely within the logic. We assume that there are implementations of this specification available, for example, `InsertSort` : `SORTFUN`, `QuickSort` : `SORTFUN`, etc.

[2]There is an example in the appendix of Sannella et al. [1990].

2.2.3 Composition

Bearing in mind the "building blocks" defined above, we can give a new form of the main specification which makes explicit the intention to use them:

```
BUILD_KWICFUN =def
  Π Circshift : CIRCSHIFTFUN . Π Sort : SORTFUN . KWICFUN1
```

We can proceed now by what [Sokołowski, 1989] calls a *canonical implementation step.* This is to "follow the hint" provided by the Π-abstraction, and write a parameterised algebra by λ-abstracting over the parameters appearing in the Π-specification. So we λ-abstract over implementations of CIRCSHIFTFUN and SORTFUN, and can then implement $\mathsf{KWICFUN}_1$ in a context where they are available for use. This allows development to proceed independently and in parallel, because work on an implementation of $\mathsf{KWICFUN}_1$ can take place under the assumption that CIRCSHIFTFUN and SORTFUN have been implemented already.

```
Build_Kwic =def
  λ Circshift : CIRCSHIFTFUN
    Sort : SORTFUN .
      ...
```

2.2.4 Final step

It is fairly straightforward to implement BUILD_KWICFUN directly, making use of similar functions to those used in the specification of KWICFUN — following the bias of the specification. More efficient implementations would certainly be possible, but they would probably be harder to prove correct.

The algorithm implemented for make_kwic is:

- build the circular shifts of the keywords in each title;
- form pairs of keyword shifts and emphasised titles by emphasising the first word in each circular shift for each title;
- sort the pairs by their first component, with the lexicographic extension of Word.≤

The output is the list formed by taking all the second components of the final list. The result looks something like this:

```
Build_Kwic =def
  λ Circshift : CIRCSHIFTFUN
    Sort : SORTFUN
```

```
List : LISTFUN
Word : WORD₁
Keyword : KEYWORD₁(Word)
EmphWord : EMPHWORD₁(Word)
Title : TITLE₁ (Word, List (alg type elt = Word.word end))
EmphTitle :
  EMPHTITLE₁ (Word, EmphWord,
             List (alg type elt = EmphWord.word end))
 TitleList : LIST (spec type elt = Title.title end)
 EmphTitleList : LIST (spec type elt = EmphTitle.title end) .
   local
     mod WordList = List (alg type elt = Word.word end)
     mod WordListList = List (alg
                                   type elt = Wordlist.list
                                 end)
     mod Crc = Circshift (Word, WordList, WordListList)
     type wordtitle = WordList.list × EmphTitle.title
     mod WordTitleList = List (alg elt = wordtitle end)
     mod Srt = Sort (alg elt = wordtitle
                         ≤ = Word.≤lexf ∘ fst end,
                     WordTitleList)
   in
     val make_kwic =
       fun (ts : EmphTitleList.list).
         local
           val keywords = ...
           val shifts = Crc.circ_shifts(keywords)
           val titleshifts = ...
           val sorted_titleshifts = Srt.sort(titleshifts)
         in
           map fst sorted_titleshifts
         end
     end
   end
```

This program uses some local modules and type definitions to express the algorithm described above; the function $\leq_{\text{lexf}}$ is some implementation of $\leq_{\text{lex}}$. The omitted parts of the program can be filled in fairly easily, following the specification for hints.

2.2.5 Proof of correctness

We are obliged to show that the axioms of $\mathtt{KWICFUN}_1$ are indeed satisfied in the body of `Build_Kwic`. This can be done by making use of the properties of the argument algebras `Circshift...EmphTitle`. Finally we will have established that `Build_Kwic : BUILD_KWICFUN`, and we can glue the components together to deliver the final product:

$$\mathtt{Kwic} =_{\text{def}} \mathtt{Build\_Kwic(Circshift,QuickSort)}$$

and evidence that

$$\mathtt{Kwic} : \mathtt{KWICFUN}_1 \leq \mathtt{KWICFUN}$$

Once the `Kwic` program and its correctness proof is delivered, it can be applied some algebras of the right signatures, satisfying the argument specifications, to finally get a keyword-in-context index.

2.2.6 Changing the requirements

Imagine that the program `Kwic` has been in use for some time, before the customers realise that they want to deal with indexes in which the word "and" *is* a keyword. They realise that $\mathtt{KWICFUN}_1$ ought to have been their requirement at the outset, so now they ask for an implementation of $\mathtt{KWICFUN}_1$ instead.

But the customers are reluctant to accept *any* implementation of $\mathtt{KWICFUN}_1$ now, because they have generated indexes using `Kwic` and they would prefer that any new implementation $\mathtt{Kwic}_1 : \mathtt{KWICFUN}_1$ built the same indexes as before, when given the same set of keywords. (Notice that `KWICFUN` admits some freedom: it does not specify the output order of titles whose circular-shifted keyword lists are identical.)

ASL+ has a way of specifying this requirement. The singleton specification operator is a higher-order operator, which is *tagged* with a specification that its argument must satisfy. The requirement can be posed by:

$$\mathtt{KWICFUN}_2 =_{\text{def}} \{\mathtt{Kwic}\}_{\mathtt{KWICFUN}}$$

This is a sort of abstract-model specification in a higher-order setting. Intuitively, $\{M\}_{SP}$ stands for the equivalence class of M considered at SP. The equivalence class of `Kwic` at `KWICFUN` is the collection of all functions which, when restricted to the domain of `KWICFUN`, are equal to `Kwic`. Introducing a *stratified equality* relative to a specification, written $M = N : SP$, the general rule for λ is:

$$\frac{SP' \leq SP \qquad \begin{array}{c}[X : SP'] \\ \vdots \\ M = M' : SP''\end{array}}{\lambda X : SP.M = \lambda X : SP'.M' \; : \Pi X : SP'.SP''} \quad (\text{EQ-}\lambda)$$

which allows us to consider a more permissive equality between higher-order objects, when given some specification which restricts the observations we can perform. This equality is the motivation for using a PER semantics; the details are first introduced for a subtyping system in Chapter 3.

The software company is now in the enviable position of being able to charge money for doing no work! They can easily provide an implementation of `KWICFUN`$_1$ which also satisfies `KWICFUN`$_2$ — namely `Kwic` itself, since they had the foresight to implement `KWICFUN`$_1$ anyway.

In general, implementing a singleton specification $\{M\}_{SP}$ can involve considerable work. For one thing, $\{M\}_{SP}$ is inconsistent unless $M : SP$, which may require proof. For another thing, although M will always be a permissible implementation, it may not be the best, or even a feasible one. In this case we could use `Kwic` because it also satisfies `KWICFUN`$_1$. If the software house had instead given an implementation of `KWICFUN` which relied on the specified non-keywords, then we would have to construct a new implementation which behaved as `Kwic` when given a set of old keywords, but used a different algorithm for the general case.

This example also demonstrates a use for a form of *intersection types* in ASL+, since the specification the customer wants implemented can be formally described as:

$$\mathtt{KWICFUN}_1 \bigwedge \mathtt{KWICFUN}_2$$

Semantically, intersection types are like specification unions, as used in ASL (see Section 1.2). Here we use the intersection operator more generally than in ASL, since we are combining two specifications of parameterised programs; collections of functions rather than collections of algebras. Specification unions were added to the higher-order part of ASL+ in Sannella et al. [1992], but they will not be treated in such generality in this thesis (see comments in Section 5.2).

2.3 Summary

In this chapter I sketched the use of ASL+ in the process of formal program development. The keyword-in-context example is somewhat small and contrived, and most of the modules are abstract data types. More examples of the higher-order parameterisation facilities given by ASL+ should be studied, although getting the familiar part of the calculus right is a sensible first goal.

Some of the details of the example were omitted, or were hazy. Part of the quest of the following chapters is to demonstrate that this example can be given a completely formal underpinning.

3 Singleton Types

A typed λ-calculus called $\lambda_{\leq\{\}}$ is introduced, which combines subtypes and *singleton types.* The calculus is a minimal calculus of subtyping with a restricted form of dependent types.

The presence of dependent types leads to a circularity between the definitions of the judgements, which means that meta-theoretic properties are more difficult to establish than for systems without subtyping or without dependent types. This circularity is tackled here to prove that the system has some good meta-theoretic properties, including *subject reduction* and *minimal types.*

A PER model for $\lambda_{\leq\{\}}$ is defined, combining previous work on PER models for non-dependent subtyping systems and for dependent systems without subtyping. This is the first PER model for a system with both features.

3.1 Introducing Singletons

TYPE SYSTEMS for current programming languages provide only coarse distinctions amongst data values: *real*, *bool*, *string*, etc. Constructive type theories for program specification can provide very fine distinctions such as $\{ x \in \mathit{nat} \mid \mathit{prime}(x) \}$, but the problem of checking whether a term inhabits a type may be undecidable. We need to study systems in between, so that types may express more exact requirements in programs, without losing the possibility of efficient mechanical type-checking.

Intersection types are one way of making types more exact, constructing types by "cutting down from above." By contrast, *singleton types* allow types to be "built from below."

If we view types as specifications, then a singleton type imposes the most stringent requirement imaginable. Let *fac* stand for the program

$$\mu f.\, \lambda x{:}\mathit{nat}.\, \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * (f(x-1))$$

Then $\{\mathit{fac}\}$ is a specification of the factorial function, and

$$\mathit{fac} : \{\mathit{fac}\}$$

says that *fac* certainly satisfies the specification $\{\mathit{fac}\}$. This is an instance of the principal assertion for singleton types, $M : \{M\}$.

But we can write the factorial function in many other ways; it would be useful if whenever *fac'* is an implementation of the factorial function, we have also:

$$\mathit{fac}' : \{\mathit{fac}\}.$$

This leads to the idea of letting $\{M\}$ stand for the collection of terms that have the same denotation as M in the semantics, or as an approximation, the collection of terms equal to M in some theory of equality.

Subtyping systems can substitute a term of a desired type with a term having a more refined type; the principal rule for subtyping is *subsumption*:

$$\frac{M : A \qquad A \le B}{M : B}$$

Subsumption for ground types suggests subtyping at higher types; for example, a function defined over *int* may be used where one defined only over *nat* is needed, since every natural is an integer and thus a valid argument. So we expect $\mathit{int} \to \mathit{int}$ to be a subtype of $\mathit{nat} \to \mathit{int}$.

A *stratified equality* now arises: we may have two functions defined over *int* that have equal values at every natural:

$$(\lambda x{:}\mathit{int}.\, \textbf{if } x > 0 \textbf{ then } x \textbf{ else } 2 * x) \;=\; (\lambda x{:}\mathit{int}.\, x) \;\; : \mathit{nat} \to \mathit{int}$$

but can be differentiated on integers:

$$(\lambda x{:}\mathit{int}.\, \textbf{if } x > 0 \textbf{ then } x \textbf{ else } 2 * x) \;\neq\; (\lambda x{:}\mathit{int}.\, x) \;\; : \mathit{int} \to \mathit{int}$$

These functions are interchangeable in a context where arguments of type *nat* are supplied.

These two ideas influence my treatment of singleton types. We shall consider $\{M\}$ to be an equivalence class of terms, but relative to a particular type. So a type-tag is attached to the singleton, and the introduction rule for singletons is:

$$\frac{M : A}{M : \{M\}_A}$$

Singleton types have a *non-informative* flavour, that is to say, there is no term operator corresponding to singleton introduction, as would be typical for types in a constructive type theory.

Neither is there an elimination operator for singleton types. In fact, no explicit elimination rules for singleton types will be used at all. Instead, we treat singleton types as a way of expressing the natural typed-equational theory of the system, which would otherwise be given as an additional collection of rules. The typing assertion $M : \{N\}_A$ asserts $M = N : A$. This results in a more perspicuous system than using a rule of untyped β-conversion does. In particular, it can be given a semantical interpretation directly without relying on meta-properties such as the Church-Rosser property and subject reduction. Further discussion on this point is given in the next chapter, in Section 4.4.2.

The non-informative aspect of singleton types makes the meta-theory of the system harder to deal with. Because elements of singleton types are not distinguished from other types by constructive information, the uniqueness of types property is lost. This is in contrast to elements of equality types, for example. Subtyping itself already breaks uniqueness of types, of course, so the key to solving the meta-theory is to understand the interaction between subtyping and singleton types.

In the rest of the chapter I study the addition of singleton types to the simply-typed λ-calculus with subtyping, known as $\lambda_{\leq}$. The resulting system, $\lambda_{\leq\{\}}$, is a minimal type system with singleton types and subtyping. Singleton types introduce type-term dependency, which complicates the study, but leads to a system with some interesting aspects, such as the integration of definitions and the ability to "escape" λ-abstraction and type function bodies in the presence of their arguments.

The system $\lambda_{\leq\{\}}$ is original,[1] although it is related to a fragment of the system ATTT due to Hayashi [1994]; my singleton types have the same form as Hayashi's, but are handled in a different way, see Section 3.7 for a comparison. As far as I am aware, Hayashi's systems and the work that spawned $\lambda_{\leq\{\}}$ in Sannella et al. [1992] are the only other appearances of singleton types in the literature.

[1]Most of the content of this chapter was first published in the paper "Subtyping with Singleton Types" which I presented at *Computer Science Logic 1994* [Aspinall, 1995a].

The next section outlines some possible applications of type systems with singleton types. In Section 3.3 the complete definition of $\lambda_{\leq\{\}}$ is presented and in the two following sections, some properties of the syntax are established. In Section 3.6 $\lambda_{\leq\{\}}$ is given a PER semantics and a proof of soundness. Section 3.7 has some discussion and comparison to related work, and Section 3.8 concludes with a summary.

This chapter and Chapter 4 can each be read independently of the rest of the thesis. I have tried to make this chapter accessible to readers without a background in type-theory or subtyping, and I have tried to make it serve as an introduction to the work later. Nonetheless, a standard reference for typed lambda calculi may be helpful, for example Barendregt [1992].

3.2 Uses of Singleton Types

The original motivation for this work comes from types-as-specifications: given a program P we can form the very tight specification $\{P\}$ which is met uniquely by P. However, singleton types may have other uses inside a type system.

3.2.1 Singleton types as definitions

Definition by abbreviation is essential for the practical use of a type system in a programming language or proof assistant. If M is a large expression occurring inside N several times, we may write

$$x = M \text{ in } N'$$

instead, where N' is the result of replacing occurrences of M in N with the variable x. Treatments in the literature include Harper and Pollack [1991] and Severi and Poll [1994]. Typically, definitions are introduced as a new concept that extends the type theory being studied and causes additional complication; with singleton types we get a form of definitions in the system for free.

Severi and Poll [1994] study the addition of definitions to Pure Type Systems [Barendregt, 1992]. The setting is Church-style, so typed definitions are appropriate, with the form

$$x = M : A \text{ in } N$$

This may be compared to a λ-abstraction over a singleton type in $\lambda_{\leq\{\}}$ which is applied to a trivially suitable argument:

$$(\lambda x{:}\{M\}_A.\, N)\, M$$

This in turn can be compared with the regular "trick" in $\lambda^{\rightarrow}$:

$$(\lambda x{:}A.\,N)\,M$$

Severi and Poll point out three reasons for introducing definitions as a new concept:

1. β-reduction replaces all instances of x in N by M, whereas it is useful to be able to replace instances one-by-one when desired.

2. The information that $x = M$ may lead to a (different) typing for N that otherwise wouldn't be possible.

3. The λ-abstraction $\lambda x : A.N$ may not be permitted in the type-system.

The third point is relevant in the context of Pure Type Systems. To deal with the first point, Severi and Poll introduce a new kind of reduction. A δ-reduction replaces a single instance of x with M. We might equally well consider a new kind of reduction in $\lambda_{\leq\{\}}$, defined for applications having the special form shown above; it isn't necessary to add a new syntactic form to do this. Reduction relations aren't studied here, but it would be an interesting extension.

In $\lambda_{\leq\{\}}$ we have the benefit of the second point. Interestingly, because of the presence of singleton types, it isn't even necessary for the term M to be "revealed" to the function body N in the type label to use $x = M$ when typing N. The term $(\lambda x{:}A.\,N)\,M$ has exactly the same types as $(\lambda x{:}\{M\}_A.\,N)\,M$. Moreover the two terms are provably equal in $\lambda_{\leq\{\}}$.

3.2.2 Singleton types and improved typings

The system $\lambda_{\leq\{\}}$ is a non-conservative extension of the well-known subtyping systems; better non-dependent typings are possible than in non-dependent subtyping systems. A simple concrete example is the identity function on real numbers, written $\lambda x{:}\mathit{real}.\,x$. Then in $\lambda_{\leq}$:

$$\begin{array}{rcl} \lambda x{:}\mathit{real}.\,x & : & \mathit{real} \rightarrow \mathit{real} \\ & : & \mathit{int} \rightarrow \mathit{real} \\ & \not: & \mathit{int} \rightarrow \mathit{int} \end{array}$$

The third typing is possible in $\lambda_{\leq\{\}}$, so a given $\lambda_{\leq}$ function can have more $\lambda_{\leq}$ typings in $\lambda_{\leq\{\}}$.

In a programming setting, singletons could be used to derive good typings between a collection of functions inside a module, and then by removing detail from these types we could get a collection of non-dependent types for typing outwith the module boundary. The example in Section 3.2.1 shows how we can "in-line" function arguments when type-checking, which is sure to result in better typings than a more abstract non-dependent scheme (such as intersection types) would do alone. But I haven't examined examples in detail, so it remains to find out how useful this could be.

The difficulty, of course, is in knowing when to in-line and how to manage the complexity of type-checking. Dependent types will lead to an undecidable type system unless there is some decidable restriction on the form of terms allowed in types, so that equality can be tested. In practice it might be necessary to annotate programs with directives to indicate to the compiler when to use extra-informative, dependent, typings.

One possibly desirable theoretical property is missed, though. In general, there may be no minimal type in $\lambda_{\leq\{\}}$ amongst the possible $\lambda_{\leq}$ types for a given term. This can be seen from the identity function example above: $\mathit{real} \to \mathit{real}$ and $\mathit{int} \to \mathit{int}$ are incomparable types. One way to repair this would be to add intersection types. However, $\lambda_{\leq\{\}}$ itself *does* possess the minimal type property, see Section 3.5.3.

3.3 The System $\lambda_{\leq\{\}}$

Now we describe the system $\lambda_{\leq\{\}}$, summarized in the tables at the end of the chapter beginning on page 80. The presentation follows the familiar format for subtyping systems: we have well-formedness, typing, and subtyping judgements. Additionally, we have the pure equational theory generated by the singleton rules as explained in the previous section.

Syntax. The meta-variables $A, B, C \ldots$ and $M, N, \ldots$ and Γ range over pre-types, pre-terms and pre-contexts respectively, which are given by the grammar:

$$\begin{array}{rcl} A & ::= & \kappa \mid \Pi x{:}A.B \mid \{M\}_A \\ M & ::= & x \mid \lambda x{:}A.M \mid MN \\ \Gamma & ::= & \langle\rangle \mid \Gamma, x : A \end{array}$$

Since there are no type variables, we assume the existence of a set of atomic type symbols $\mathcal{K}$, ranged over by κ. Constants are not included in the system, for the sake of brevity; we can simulate them by using a fixed initial context.

I shall follow the usual conventions and use obvious abbreviations in what follows. Free and bound variables are defined as usual. Pre-types, pre-terms, and pre-contexts that are alpha-convertible are identified, and I write $\equiv$ to stand for syntactic identity. Pre-contexts are additionally restricted so that no variable x is declared more than once; this is assumed implicitly in the typing rules.

Beta-reduction and conversion are defined as usual over terms, and extended to types in a way compatible with (i.e. as a congruence with respect to) the type-constructors. For notation, $M \longrightarrow_\beta N$ indicates an outermost single-step of reduction, and $M \twoheadrightarrow_\beta N'$ is the transitive, reflexive, and compatible closure of $M \longrightarrow_\beta N$. Beta-conversion is the symmetric closure of $\twoheadrightarrow_\beta$, written $M =_\beta N$. There is no application at the level of types; convertible types can only differ within corresponding singleton components.

Judgements. The judgement forms are:

$\rhd \Gamma$	Γ is a well-formed context
$\Gamma \rhd A$ *type*	A is a well-formed type in Γ
$\Gamma \rhd M : A$	M has type A in Γ
$\Gamma \rhd A \leq B$	A is a subtype of B in Γ

I occasionally use $\Gamma \rhd J$ to range over judgements with context Γ. A judgement is *valid* iff it can be derived using the rules of Figures 3.1–3.4, shown at the end of the chapter, beginning on page 80. As usual, "$\Gamma \rhd J$" abbreviates "$\Gamma \rhd J$ is valid." The rules are described individually in the next subsection.

There is an auxiliary notation for the equality judgement defined in terms of the typing judgement:

$$\Gamma \rhd M = N : A \qquad =_{\text{def}} \qquad \Gamma \rhd M : \{N\}_A$$

The four primitive judgements must be defined simultaneously: it is characteristic that typing should be dependent on the formation judgements through (VAR) and subtyping through (SUB). The presence of dependent types means that formation and subtyping are also dependent on the typing judgement, through rules (FORM-$\{\}$), (SUB-$\{\}$), (SUB-EQ-SYM), and (SUB-EQ-ITER).

3.3.1 *Rules defining* $\lambda_{\leq\{\}}$

Here is an explanation of the rules defining the system.

Contexts

There are just two rules of context formation,

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

$$\frac{\Gamma \triangleright A\ type}{\triangleright \Gamma, x : A} \qquad \text{(EXTEND)}$$

and the rule of variable typing makes use of the context:

$$\frac{\triangleright \Gamma}{\Gamma \triangleright x : \Gamma(x)} \qquad \text{(VAR)}$$

Basic subtyping rules

The rule of subsumption is the characteristic rule of type systems with subtyping:

$$\frac{\Gamma \triangleright M : A \qquad \Gamma \triangleright A \leq B}{\Gamma \triangleright M : B} \qquad \text{(SUB)}$$

It captures the informal meaning of the subtype relation: if type A is a subtype of type B, then every element of A is also an element of B.

The subtyping relation is reflexive on well-formed types:

$$\frac{\Gamma \triangleright A\ type}{\Gamma \triangleright A \leq A} \qquad \text{(SUB-REFL)}$$

and transitive:

$$\frac{\Gamma \triangleright A \leq B \qquad \Gamma \triangleright B \leq C}{\Gamma \triangleright A \leq C} \qquad \text{(SUB-TRANS)}$$

Atomic types

The rule (FORM-ATOMIC) says that a primitive type is valid in a valid context:

$$\frac{\triangleright \Gamma}{\Gamma \triangleright \kappa\ type} \qquad \text{(FORM-ATOMIC)}$$

Subtyping between atomic types is given by a built-in relation,

$$\leq_{Atom} \ \subseteq \ \mathcal{K} \times \mathcal{K}$$

which we assume to be reflexive and transitive. Restricting $\leq_{Atom}$ to atomic types ensures that subtyping retains a *structural* character; that is, types related by the subtype relation will have a similar shape.

The rule (SUB-ATOMIC) includes $\leq_{Atom}$ in the subtyping relation:

$$\frac{\triangleright \Gamma \qquad \kappa \leq_{Atom} \kappa'}{\Gamma \triangleright \kappa \leq \kappa'} \qquad \text{(SUB-ATOMIC)}$$

Dependent product types

Dependent product types (Π-types) are formed by the rule:

$$\frac{\Gamma, x : A \rhd B \ \mathit{type}}{\Gamma \rhd \Pi x{:}A.\, B \ \mathit{type}} \qquad \text{(FORM-}\Pi\text{)}$$

Lambda abstraction introduces a term of a Π-type:

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

and function application eliminates such a term:

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

These three rules are common to all type theories with dependent products. Note that our function spaces here are strictly first order, in the sense that there is no abstraction or quantification over types.

The subtyping rule for dependent products (SUB-Π) is contravariant in the domain of the function space and covariant in the codomain:

$$\frac{\Gamma \rhd A' \le A \qquad \Gamma, x : A' \rhd B \le B' \qquad \Gamma, x : A \rhd B \ \mathit{type}}{\Gamma \rhd \Pi x{:}A.\, B \le \Pi x{:}A'.\, B'} \qquad \text{(SUB-}\Pi\text{)}$$

The contravariance is intuitive and semantically sound, but its form is sometimes problematical to the syntax, as with the corresponding form of *bounded quantification* over second order types [Ghelli, 1990, Pierce, 1992]. More investigation is needed to decide the point here: a contributing factor in the second order case is the presence of a top type $\top$ of which there is no analogue in $\lambda_{\le\{\}}$.

Singleton types and equality

Singleton types are formed by the rule:

$$\frac{\Gamma \rhd M : A \ \mathit{type}}{\Gamma \rhd \{M\}_A \ \mathit{type}} \qquad \text{(FORM-\{\})}$$

which says that $\{M\}_A$ is a valid type in some context Γ if the term M has type A in Γ. Terms of singleton type are introduced by the equality rules, principally reflexivity:

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

(this is the rule of singleton introduction from Section 3.1 under a different guise, by the definition that $M = M : A$ means $M : \{M\}_A$). Symmetry and transitivity are derived rules using the subtyping rules shown below. There are two other equality rules:

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

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

(EQ-λ) deserves some discussion. The first two premises allow one to derive equalities between functions on restricted domains, like the example shown in Section 3.1; the final premise is a well-formedness constraint, to ensure that the term $\lambda x{:}A.\, M$ appearing in the conclusion is typable. The usual form of λ-equality is between functions with the same domain,

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

It is an admissible instance of (EQ-λ). In fact, the presence of the stronger equality rule leads to the admissibility of a correspondingly stronger typing rule:

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

which allows one to give a more refined type for a function, based on more refined knowledge about its argument type. This is in contrast with usual subtyping systems where a function has a single type via λ-introduction that may be promoted via (SUB). This stronger rule was the basis of the example in Section 3.2.2. To include (EQ-λ) in the stronger form was a design decision, but it is interesting to notice that in a system with untagged singletons, the stronger rule is forced by the rule (SUB-Π).

Subtyping singletons

Subtyping of singleton types is provided by three rules. First, we propose that a singleton is a subtype of its containing type:

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A \leq A} \quad (\text{SUB-}\{\})$$

Another property we require is the principle of monotonicity of equality with respect to subtyping; if two terms are equal at a type A then they must be equal at any supertype of A:

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd M = N : B} \quad (\text{EQ-SUB})$$

We can express this principle via subtyping of singleton types. In general, as we pass from subtype to supertype, the equivalence class of any particular term gets larger:

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

Since $N : \{N\}_A$ by (EQ-REFL), assuming that $N : A$ is implied by $M = N : A$, the previous rule will follow from this one using subsumption (SUB).

Our term-level equality is reflexive (and, we will see, transitive), but not yet symmetric. We have to add a rule of symmetry, such as:

$$\frac{\Gamma \rhd M = N : A}{\Gamma \rhd N = M : A} \tag{EQ-SYM}$$

Again, we can express this rule with singleton subtyping. If we consider that two types A, B are equal if $A \leq B$ and $B \leq A$, then the rule below establishes that if two terms are equal at a type, then their equivalence classes are the same:

$$\frac{\Gamma \rhd M = N : A}{\Gamma \rhd \{N\}_A \leq \{M\}_A} \tag{SUB-EQC-SYM}$$

the other direction and the previous rule follows from this rule, again through (EQ-REFL) and (SUB).

In the definition of the system, we combine these two singleton subtyping rules into a single rule:

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

which is our second rule of singleton subtyping.

The final rule counteracts monotonicity of equality. Notice that singletons can be nested in the syntax, so we can form $\{M\}_A$, $\{M\}_{\{N\}_A}$, $\{M\}_{\{N\}_{\{P\}_A}}\cdots$. The equivalence class of M seen at type A is $\{M\}_A$. The equivalence class of N seen at $\{M\}_A$ is either empty, if $M \neq N : A$, or equal to $\{M\}_A$. In particular, $\{M\}_A = \{M\}_{\{M\}_A}$. We already have that $\{M\}_{\{M\}_A} \leq \{M\}_A$ by (SUB-{}), for the other direction we need a new rule:

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A \leq \{M\}_{\{M\}_A}} \tag{SUB-EQ-ITER}$$

3.4 Basic Properties

In this section we establish some basic expected properties of the presentation of $\lambda_{\leq\{\}}$. The first properties concern well-formedness conditions for contexts and the behaviour of variables.

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

Notation 3.1. Let $\Gamma \equiv x_1 : A_1, \ldots$ be a pre-context.

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

One derivation is "shorter" than another if it has fewer proof rules 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 $\mathbf{V}$. Substitution is extended to contexts componentwise, so that if $\Gamma \equiv x_1 : A_1, \ldots$ then $\Gamma[N/x] \equiv x_1 : A_1[N/x], x_2 : A_2[N/x], \ldots$. Substitution is also extended to judgements J componentwise.

Proposition 3.2 (Contexts).
Let $\Gamma \equiv x_1 : A_1, \ldots x_n : A_n$ be a pre-context. Then:

1. (Context 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 of contexts).
   If $\Gamma \rhd J$ where J is a formation or typing 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 $\Gamma' \rhd J$.

Proof Standard. Context formation and correctness of contexts follow by induction over the rules. To prove renaming, we use induction on the height of derivations and a quantification over all $Dom(\Gamma)$ renamings. Then renaming is used to prove thinning; in the cases for λ and Π when the context is extended in the premise, renaming is used to ensure that the thinned context can be extended without variable clashes. □

Notice that thinning encompasses both permutation of the context (re-ordering declarations) and weakening (adding a declaration to the end).

Proposition 3.3 (Substitution).
$\Gamma, x : A, \Gamma' \rhd J$ *and* $\Gamma \rhd M : A \implies \Gamma, \Gamma'[M/x] \rhd J[M/x]$.

Proposition 3.4 (Bound narrowing).
$\Gamma, x : A, \Gamma' \rhd J$ *and* $\Gamma \rhd A' : \mathit{Power}(A) \implies \Gamma, x : A', \Gamma' \rhd J$.

Proof Let y be a fresh variable. Using Proposition 3.2 (part 2, then part 1 repeatedly together with part 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 3.3, $\Gamma, y : A', \Gamma'[y/x] \rhd J[y/x]$. The result follows by renaming y back to x using Proposition 3.2(3). □

The next proposition shows implications between judgements. Some presentations of type theories simply require the consequences of these implications as extra premises in the rules to begin with (often implicitly); the approach taken here seems more satisfactory from the point of view of providing a minimal presentation, although it can make inductive proofs on derivation lengths more tricky.

Proposition 3.5 (Type correctness I).

1. $\Gamma \rhd M : A \implies \Gamma \rhd A\ \mathit{type}$
2. $\Gamma \rhd A \leq B \implies \Gamma \rhd A\ \mathit{type}$ *and* $\Gamma \rhd B\ \mathit{type}$

Proof Simultaneous induction on derivation heights. We show part 1 by induction over the typing rules.

Case (VAR): By (GEN-TYPE).

Case (λ): By IH and (FORM-Π).

Case (APP): IH gives $\Gamma \rhd \Pi x{:}A.\,B\ \mathit{type}$ which must have been derived with (FORM-Π) whose premise is $\Gamma, x : A \rhd B\ \mathit{type}$. Hence $\Gamma \rhd B[N/x]\ \mathit{type}$ using the premise of the rule and substitution.

Case (SUB): By IH (ii).

Case (EQ-REFL): Use (FORM-$\{\}$) on the premise.

Case (EQ-λ): Using (FORM-λ), we must show $\Gamma \rhd \lambda x{:}A'.\,M' : \Pi x{:}A.\,B'$. This follows by the induction hypothesis for the second premise and the premise of (FORM-$\{\}$).

Case (EQ-APP): Similarly.

We show part 2 by induction over the subtyping rules.

Case (SUB-REFL): Premise.

Case (SUB-TRANS): By the induction hypothesis.

Case (SUB-ATOMIC): Using (FORM-ATOMIC).

Case (SUB-Π): For the left-hand type, use the third premise and (FORM-Π). For the right-hand type, use the IH to get $\Gamma, x : A' \rhd B'$ *type* then (FORM-Π).

Case (SUB-{}): Use (FORM-{}) or IH for part 1.

Case (SUB-EQ-SYM): Use IH for part 1 to get $\Gamma \rhd \{N\}_A$, hence well-formedness of the left-hand type and also $\Gamma \rhd N : A$. Then:

$$\dfrac{\dfrac{\Gamma \rhd M : \{N\}_A \qquad \dfrac{\Gamma \rhd N : A}{\Gamma \rhd \{N\}_A \leq A} \qquad \Gamma \rhd A \leq B}{\Gamma \rhd M : B}}{\Gamma \rhd \{M\}_B \textit{ type}}$$

which shows well-formedness of the right-hand type.

Case (SUB-EQ-ITER): Use premise and (FORM-{}) for the left-hand type; for the other side use (EQ-REFL) then (FORM-{}). □

A *generation principle* decomposes a derived judgement into further derived judgements, usually subderivations of the first. It tells how a particular judgement was generated.

For the context and type-formation judgement, the generation principles are merely inversions of the rules; there is at most one rule that could have been used last in the derivation of a given type-formation judgement. This is used to show a simple corollary of the above proposition.

Corollary 3.6 (Type correctness II).

3. $\Gamma \rhd M = N : A \quad \Longrightarrow \quad \Gamma \rhd M : A$

4. $\Gamma \rhd M = N : A \quad \Longrightarrow \quad \Gamma \rhd N : A$

5. $\Gamma \rhd \{M\}_A \leq B \quad \Longrightarrow \quad \Gamma \rhd M : A$ *and* $\Gamma \rhd M : B$

6. $\Gamma \rhd \{M\}_A \leq B \quad \Longrightarrow \quad \Gamma \rhd M : B$

Proof Parts 4 and 5 follow from part 1 and part 2 respectively, by the premise of (FORM-{}). Parts 3 and 6 follow from part 4 and 5 respectively, using (SUB-{}) and (EQ-REFL), with (SUB). □

We now give generation principles for the subtyping and typing judgements. First showing a generation result for the subtyping judgement allows us to then prove one for the typing judgement. There is a case according to each syntactic form on either side of the subtyping symbol.

Proposition 3.7 (Subtyping generation).

1. $\Gamma \rhd \kappa \leq B \implies$ *For some* κ',
   $B \equiv \kappa'$
   $\kappa \leq_{Atom} \kappa'$

2. $\Gamma \rhd \Pi x{:}A.\, B \leq C \implies$ *For some* A',B',
   $C \equiv \Pi x{:}A'.\, B'$,
   $\Gamma \rhd A' \leq A$,
   $\Gamma, x : A' \rhd B \leq B'$
   $\Gamma, x : A \rhd B$ *type*.

3. $\Gamma \rhd \{M\}_A \leq B \implies \Gamma \rhd M : B$

4. $\Gamma \rhd A \leq \kappa' \implies$ *For some* κ,
   where $A \not\equiv \{M\}_B$ $\quad A \equiv \kappa$
   $\kappa \leq_{Atom} \kappa'$

5. $\Gamma \rhd C \leq \Pi x{:}A'.\, B' \implies$ *For some* A,B,
   where $C \not\equiv \{M\}_D$ $\quad C \equiv \Pi x{:}A.\, B$
   $\Gamma \rhd A' \leq A$
   $\Gamma, x : A' \rhd B \leq B'$
   $\Gamma, x : A \rhd B$ *type*.

6. $\Gamma \rhd C \leq \{N\}_B \implies$ *For some* $M, A, C \equiv \{M\}_A$

Proof We use induction on the derivation of subtyping judgements to show parts part 1 and part 2; only the possible last rules are considered in the cases below. Part 3 follows from Corollary 3.6 and (SUB-{}). Parts 4–6 then follow directly by consideration of parts 1–3.

1. **Case** (SUB-REFL): Immediate, by reflexivity of $\leq_{Atom}$.

   Case (SUB-TRANS): By the induction hypothesis for the left premise, then the right, then the result by transitivity of $\leq_{Atom}$.

   Case (SUB-ATOMIC): Immediate.

2. **Case** (SUB-REFL): By Proposition 3.5 and type-formation generation, $\Gamma, x : A \rhd B$ *type* and by Proposition 3.2, (GEN-TYPE), $\Gamma \rhd A$ *type*.

   Case (SUB-TRANS): By the induction hypothesis for the left premise, then the right premise, and the result using (SUB-TRANS) again.

   Case (SUB-Π): Immediate by the premises. □

To show a generation principle for typing, we make use of some admissible rules, taken from Figure 3.5 on page 83. The admissible rules are described in Section 3.4.1.

The generation principle for the typing judgement $\Gamma \rhd M : A$ looks unusual, because we must account for the possibility that A is a singleton.

Proposition 3.8 (Typing generation).

1. $\Gamma \rhd x : A \implies \Gamma \rhd \{x\}_{\Gamma(x)} \leq A$

2. $\Gamma \rhd \lambda x{:}A.\, M : C \implies$ *For some* A', B, B',
$$\begin{array}{l} \Gamma \rhd A' \leq A \\ \Gamma, x : A' \rhd M : B' \\ \Gamma, x : A \rhd M : B \\ \Gamma \rhd \{\lambda x{:}A.\, M\}_{\Pi x:A'.\, B'} \leq C \end{array}$$

3. $\Gamma \rhd MN : C \implies$ *For some* A, B,
$$\begin{array}{l} \Gamma \rhd M : \Pi x{:}A.\, B \\ \Gamma \rhd N : A \\ \Gamma \rhd \{MN\}_{B[N/x]} \leq C \end{array}$$

Proof We make use of this admissible rule:

$$\frac{\Gamma \rhd \{M\}_A \leq B}{\Gamma \rhd \{M\}_A \leq \{M\}_B} \qquad \text{(SUB-INCL)}$$

which is derived thus:

$$\frac{\dfrac{\dfrac{}{\Gamma \rhd M : A}}{\Gamma \rhd \{M\}_A \leq \{M\}_{\{M\}_A}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \rhd M : A}}{\Gamma \rhd M = M : A} \qquad \Gamma \rhd \{M\}_A \leq B}{\Gamma \rhd \{M\}_{\{M\}_A} \leq \{M\}_B}}{\Gamma \rhd \{M\}_A \leq \{M\}_B}$$

using (SUB-EQ-ITER), (SUB-EQ-SYM) and Proposition 3.5. Each case of the proposition 3.8 is proved by induction on the derivation.

1. **Case** (VAR): then $\Gamma \rhd \{x\}_{\Gamma(x)} \leq \Gamma(x)$ by (SUB-{}).

   Case (SUB): by IH, (SUB-TRANS).

   Case (EQ-REFL): let the premise be $\Gamma \rhd x : A$.
   By the IH, $\Gamma \rhd \{x\}_{\Gamma(x)} \leq A$. Hence the result using (SUB-INCL).

2. **Case** (λ): we have $\Gamma, x : A \rhd M : B$ by the premise, and
   $\Gamma \rhd \{\lambda x{:}A.\, M\}_{\Pi x:A.\, B} \leq \Pi x{:}A.\, B$ by (SUB-{}).

   Case (SUB): by IH, (SUB-TRANS).

   Case (EQ-REFL): by IH, (SUB-INCL).

Case (EQ-λ): by the premises, $\Gamma, x : A \rhd M : B$ and $\Gamma, x : A' \rhd M : B'$, using Corollary 3.6.
$\Gamma \rhd \{\lambda x{:}A.\,M\}_{\Pi x:A'.B'} \le \{\lambda x{:}A'.\,M'\}_{\Pi x:A'.B'}$ follows by (SUB-EQ) (using (SUB-REFL) and Proposition 3.5).

3. **Case** (APP): by premises and (SUB-{}).

   Case (SUB): by IH, (SUB-TRANS).

   Case (EQ-REFL): by IH, (SUB-INCL).

   Case (EQ-APP): by Corollary 3.6, $\Gamma \rhd M : \Pi x{:}A.\,B$ and $\Gamma \rhd N : A$. The result by (SUB-EQ) as above. □

The consequence of typing generation can be further broken down in specific instances by using the subtyping generation principle once more, and so on.

3.4.1 *Admissible rules of* $\lambda_{\le\{\}}$

We continue the development of the meta-theory by showing some important admissible rules of $\lambda_{\le\{\}}$ in Table 3.5, several of which have been mentioned before. They are important either because it is natural to want them to hold, or because they are useful in following proofs.

First, we have symmetry and transitivity of equality:

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

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

Interestingly, the usual rule for β-conversion turns out to be admissible. We can give $\lambda x{:}A.\,M$ the tight dependent type $\Pi x{:}A.\,\{M\}_B$ using ({}-I), and then use (APP) to give:

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

There are several easily derived rules for subtyping singletons. One is (SUB-EQ), recovered from (SUB-EQ-SYM):

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \le B}{\Gamma \rhd \{M\}_A \le \{N\}_B} \qquad \text{(SUB-EQ)}$$

An important instance of this rule is when $A \equiv B$, and the second premise is implied by the first, by Corollary 3.6.

Corresponding to (SUB-EQ), (EQ-SUB) shows the monotonicity of equality wrt the subtyping relation.

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

We mentioned the final rule in the table, (λ-NARROW), in Section 3.3.1.

Proof of admissible and derivable rules We prove the admissibility or derivability of the rules in Figure 3.5.

- (λ-NARROW) holds because we may derive:

$$\frac{\Gamma \rhd A' \leq A \qquad \dfrac{\Gamma, x : A' \rhd M : B'}{\Gamma, x : A' \rhd M = M' : B'} \qquad \Gamma, x : A \rhd M : B}{\Gamma \rhd \lambda x{:}A.\, M = \lambda x{:}A'.\, M : \Pi x{:}A'.\, B'}$$

  and

$$\frac{\dfrac{\Gamma, x : A' \rhd M : B'}{\Gamma \rhd \lambda x{:}A'.\, M : \Pi x{:}A'.\, B'}}{\Gamma \rhd \{\lambda x{:}A'.\, M\}_{\Pi x:A'.\, B'} \leq \Pi x{:}A'.\, B'}$$

  and then use (SUB) to get $\Gamma \rhd \lambda x{:}A.\, M : \Pi x{:}A'.\, B'$.

- (EQ-SYM) By this derivation, using Proposition 3.5.

$$\frac{\dfrac{\Gamma \rhd N : A}{\Gamma \rhd N = N : A} \qquad \dfrac{\Gamma \rhd M = N : A \qquad \dfrac{\Gamma \rhd A \textit{ type}}{\Gamma \rhd A \leq A}}{\Gamma \rhd \{N\}_A \leq \{M\}_A}}{\Gamma \rhd N = M : A}$$

- (EQ-TRANS) Using (EQ-SYM) and Proposition 3.5 again:

$$\frac{\Gamma \rhd L = M : A \qquad \dfrac{\dfrac{\Gamma \rhd M = N : A}{\Gamma \rhd N = M : A} \qquad \dfrac{\Gamma \rhd A \textit{ type}}{\Gamma \rhd A \leq A}}{\Gamma \rhd \{M\}_A \leq \{N\}_A}}{\Gamma \rhd L = N : A}$$

- (EQ-SUB) Follows easily from (EQ-SYM), (SUB), (SUB-EQ-SYM).

- (EQ-β) Using (λ) and (EQ-APP), we have:

$$\frac{\dfrac{\dfrac{\Gamma, x : A \rhd M : B}{\Gamma, x : A \rhd M : \{M\}_B}}{\Gamma \rhd \lambda x{:}A.\, M : \Pi x{:}A.\, \{M\}_B} \qquad \Gamma \rhd N : A}{\Gamma \rhd (\lambda x{:}A.\, M)\, N = M[N/x] : B[N/x]}$$

- (SUB-EQ) by (EQ-SYM) and (SUB-EQ-SYM):

$$\frac{\dfrac{\Gamma \rhd M = N : A}{\Gamma \rhd N = M : A} \qquad \Gamma \rhd A \leq B}{\Gamma \rhd \{M\}_A \leq \{N\}_B}$$

□

3.5 Further Properties

In this section we study some further meta-theory of $\lambda_{\leq\{\}}$, working towards a proof of subject reduction and the minimal type property.

3.5.1 Removing singletons

Subtyping generation, Proposition 3.7, is rather weak in the singleton case $\{M\}_A \leq B$. We'd like to say something about the relationship between A and B; when B doesn't have the form of a singleton, we expect that $A \leq B$. However, when $B \equiv \{N\}_C$ for some C, we may have $A \leq C$ or vice-versa, because of rules (SUB-EQ-SYM) and (SUB-EQ-ITER). A generation lemma covering these cases becomes untidy to state, and difficult to prove directly because of the rule (SUB-TRANS).

Here we define an operation $(-)^{\emptyset}$ which erases outermost singletons from a type. A simple lemma relates a type to its singleton-deleted form; this strengthens the generation result sufficiently to give us a tool to help show that $\lambda_{\leq\{\}}$ possesses minimal types.

Definition 3.9 (Singleton removal).

$$\begin{aligned}
(\{M\}_A)^{\emptyset} &= A^{\emptyset} \\
P^{\emptyset} &= P \\
(\Pi x{:}A.\,B)^{\emptyset} &= \Pi x{:}A.\,B
\end{aligned}$$

Proposition 3.10 (Properties of singleton removal).

1. $\Gamma \rhd A$ *type* $\Longrightarrow \Gamma \rhd A \leq A^{\emptyset}$

2. $\Gamma \rhd A \leq B \Longrightarrow \Gamma \rhd A^{\emptyset} \leq B^{\emptyset}$

3. $\Gamma \rhd \{M\}_A \leq B$ *and* $B \not\equiv \{N\}_C \Longrightarrow \Gamma \rhd A \leq B$

Proof Part 1 by induction on the structure of types. Use (SUB-REFL) except in the case that $A \equiv \{M\}_B$ for some M, B, when we apply the induction hypothesis to obtain $\Gamma \rhd B \leq B^{\emptyset}$, and by type-formation and (SUB-$\{\}$), $\Gamma \rhd \{M\}_B \leq B$; hence the result via (SUB-TRANS).

Part 2 by induction on the subtyping derivation. For (REFL), (SUB-$\{\}$) and (SUB-EQ-ITER) we can use part 1 and Proposition 3.5. Use the induction hypothesis for (SUB-TRANS) and (SUB-EQ-SYM). There's nothing to do for (SUB-ATOMIC) and (SUB-Π).

Part 3 then follows immediately. □

3.5.2 Subject reduction

We can now use the generation principles to show that subject reduction for β holds, for both typing and subtyping. The syntactic proof of this is surprisingly involved, compared to subtyping systems without dependent types, or systems of dependent types without subtyping.

Theorem 3.11 (Subject reduction holds for $\lambda_{\leq\{\}}$).

1. *If $\Gamma \vartriangleright M : A$ and $M \longrightarrow_\beta M'$, then $\Gamma \vartriangleright M' : A$.*
2. *If $\Gamma \vartriangleright A \leq B$ and $A \longrightarrow_\beta A'$, then $\Gamma \vartriangleright A' \leq B$.*
3. *If $\Gamma \vartriangleright A \leq B$ and $B \longrightarrow_\beta B'$, then $\Gamma \vartriangleright A \leq B'$.*

Proof Simultaneously for a single reduction step, by induction on the structure of terms and types. For terms, this involves the use of Proposition 3.8 and the equality rules, plus Lemma 3.12 below. For types, we use Proposition 3.7 and Proposition 3.5. □

The critical lemma is the case of a one-step outermost reduction.

Lemma 3.12.

$$\frac{\Gamma \vartriangleright (\lambda x{:}A.\, M)\, N : C}{\Gamma \vartriangleright M[N/x] : C}$$

Proof By typing generation, Proposition 3.8, we have:

$$\begin{array}{cr}
\Gamma \vartriangleright \lambda x{:}A.\, M : \Pi x{:}A_1.\, B_1 & \\
\Gamma \vartriangleright N : A_1 & \\
\Gamma \vartriangleright \{(\lambda x{:}A.\, M)(N)\}_{B_1[N/x]} \leq C & (\star) \\
\Gamma \vartriangleright A_2 \leq A & \\
\Gamma, x : A \vartriangleright M : B & \\
\Gamma, x : A_2 \vartriangleright M : B_2 & \\
\Gamma \vartriangleright \{\lambda x{:}A.\, M\}_{\Pi x:A_2.\, B_2} \leq \Pi x{:}A_1.\, B_1 &
\end{array}$$

By Propositions 3.10,3.7 and the last of these, we have:

$$\begin{array}{c}
\Gamma \vartriangleright \Pi x{:}A_2.\, B_2 \leq \Pi x{:}A_1.\, B_1 \\
\Gamma \vartriangleright A_1 \leq A_2 \\
\Gamma, x : A_1 \vartriangleright B_2 \leq B_1
\end{array}$$

Now using bound narrowing and (SUB) we have:

$$\Gamma, x : A_1 \vartriangleright M : B_1$$

So we can apply the admissible rule (EQ-β),

$$\frac{\Gamma, x : A_1 \rhd M : B_1 \qquad \Gamma \rhd N : A_1}{\Gamma \rhd (\lambda x{:}A_1.\,M)(N) = M[N/x] \;: B_1[N/x]}$$

and by (EQ-λ), (EQ-APP):

$$\frac{\Gamma \rhd A_1 \leq A \qquad \Gamma, x : A_1 \rhd M = M : B_1 \qquad \Gamma, x : A \rhd M : B \qquad \Gamma \rhd N : A_1}{\Gamma \rhd (\lambda x{:}A.\,M)(N) = (\lambda x{:}A_1.\,M)(N) \;: B_1[N/x]}$$

By transitivity:

$$\Gamma \rhd (\lambda x{:}A.\,M)(N) \;=\; M[N/x] \;: B_1[N/x] \qquad (\star\star)$$

Finally, making use of a rule admissible via Corollary 3.6, (SUB-TRANS) and (SUB-EQ-SYM):

$$\frac{\dfrac{\Gamma \rhd P = Q : D \qquad \Gamma \rhd \{P\}_D \leq C}{\Gamma \rhd \{Q\}_D \leq \{P\}_D}}{\Gamma \rhd Q : C}$$

with $(\star\star)$ and $(\star)$ as the premises; $P \equiv (\lambda x{:}A.\,M)(N)$, $Q \equiv M[N/x]$, and $D \equiv B_1[N/x]$. Thus

$$\Gamma \rhd M[N/x] : C$$

as required. □

3.5.3 Minimal types

With untagged singletons, minimal types are a triviality: the minimal type for a term M is $\{M\}$! When type tags are added, the issue is not so obvious. Here we show a strengthening of Typing Generation to give minimal types.

The minimal type $min_\Gamma(M)$, of a term M in a context Γ, has the form $\{M\}_{A_m}$ for some A_m which we call the *non-singleton minimal type* of M. Here we give a partial inductive definition of $min_\Gamma(M)$ which we show in the following lemma to be well defined on all Γ, M such that $\Gamma \rhd M : A$ for some A.

Definition 3.13 (Minimal types).

$$\begin{aligned}
min_\Gamma(x) &= \{x\}_{\Gamma(x)} \\
min_\Gamma(\lambda x{:}A.\,M) &= \{\lambda x{:}A.\,M\}_{\Pi x:A.\,min_{\Gamma,x:A}(M)} \\
min_\Gamma(MN) &= \{MN\}_{B_m[N/x]} \quad \text{where} \quad (min_\Gamma(M))^{\emptyset} \equiv \Pi x{:}A_m.\,B_m \\
&\qquad\qquad\qquad\qquad\quad \text{and } \Gamma \rhd N : A_m
\end{aligned}$$

This lemma establishes the existence and minimality of $min_\Gamma(M)$.

Theorem 3.14 ($\lambda_{\leq\{\}}$ has minimal types).

1. $\Gamma \rhd M : A \qquad \Longrightarrow \qquad \Gamma \rhd M : min_\Gamma(M)$

2. $\Gamma \rhd M : A \qquad \Longrightarrow \qquad \Gamma \rhd min_\Gamma(M) \leq A$

Proof We prove part 1 and part 2 simultaneously by induction on the derivation of $\Gamma \rhd M : A$.

Case (VAR): part 1 by (EQ-REFL), part 2 by (SUB-{}).

Case (λ): By IH for part 1, $\Gamma, x : A \rhd M : min_{\Gamma,x:A}(M)$.
Then part 1 follows via (λ) and (EQ-REFL).
By (SUB-{}), we get $\Gamma \rhd \{\lambda x{:}A.\, M\}_{\Pi x:A.\, min_{\Gamma,x:A}(M)} \leq \Pi x{:}A.\, min_{\Gamma,x:A}(M)$ and by IH for part 2 and (SUB-Π), $\Gamma \rhd \Pi x{:}A.\, min_{\Gamma,x:A}(M) \leq \Pi x{:}A.\, B$. Hence part 2 via (SUB-TRANS).

Case (APP): By IH for part 2, $\Gamma \rhd min_\Gamma(M) \leq \Pi x{:}A.\, B$.
Using Proposition 3.10, $\Gamma \rhd min_\Gamma(M)^{\emptyset} \leq \Pi x{:}A.\, B$.
Using Proposition 3.7, we have that

$$\begin{aligned} &min_\Gamma(M)^{\emptyset} \equiv \Pi x A_m B_m \\ &\Gamma \rhd A \leq A_m \\ &\Gamma, x : A \rhd B_m \leq B \end{aligned}$$

Now $\Gamma \rhd N : A_m$ so $\Gamma \rhd MN : B_m[N/x]$ which gives part 1 using (EQ-REFL).
By substitution, Prop. 3.2, $\Gamma \rhd B_m[N/x] \leq B[N/x]$, hence part 2 using part 1, (SUB-{}) and (SUB-TRANS).

Case (SUB): part 1 by the induction hypothesis; part 2 by the induction hypothesis and (SUB-TRANS).

Case (EQ-REFL): part 1 by the induction hypothesis.
For part 2, by IH, $\Gamma \rhd min_\Gamma(M) \leq A$; we may use the admissible rule (SUB-INCL) (shown on page 60) to get $\Gamma \rhd \{M\}_{A_m} \leq \{M\}_A$.

Case (EQ-λ): part 1 by the induction hypothesis, (λ), (EQ-REFL).
part 2: by the induction hypothesis, (SUB-{}), we have $\Gamma, x : A' \rhd min_{\Gamma,x:A'}(M) \leq \{M'\}_{B'} \leq B'$ hence

$$\Gamma \rhd \Pi x{:}A'.\, min_\Gamma(M) \leq \Pi x{:}A'.\, B'$$

using (SUB-Π). By the result for part 1 and (SUB-EQ) (see below),

$$\Gamma \rhd \{\lambda x{:}A.\, M\}_{\Pi x:A.\, min_{\Gamma,x:A}(M)} \leq \{\lambda x{:}A.\, M\}_{\Pi x:A'.\, B'}$$

and by (SUB-EQ) and the conclusion of (EQ-λ),

$$\Gamma \rhd \{\lambda x{:}A.\, M\}_{\Pi x:A'.\, B'} \leq \{\lambda x{:}A'.\, M'\}_{\Pi x:A'.\, B'}$$

and so the result follows by (SUB-TRANS).

Case (EQ-APP): By IH for part 2, $\Gamma \rhd min_\Gamma(M) \leq \Pi x{:}A.B$.
Again as for (APP), we use Propositions 3.10 and 3.7 to get part 1.
For part 2, we use (SUB-EQ) with $\Gamma \rhd B_m[N/x] \leq B[N/x]$ to get $\Gamma \rhd \{MN\}_{B_m[N/x]} \leq \{MN\}_{B[N/x]}$. Again, using the original conclusion with (SUB-EQ) gives $\Gamma \rhd \{MN\}_{B[N/x]} \leq \{M'N'\}_{B[N/x]}$ and the result by (SUB-TRANS). □

3.6 A PER Interpretation of $\lambda_{\leq\{\}}$

In this section I shall define a PER model for $\lambda_{\leq\{\}}$, in a fairly direct modification of the standard definitions for $\lambda_{\leq}$ and related calculi [see for example Mitchell, 1996, Cardelli and Longo, 1991, Bruce and Longo, 1990]. We interpret types as PERs (partial equivalence relations) over a global value space D, which is the domain of a model of the untyped λ-calculus. PERs often feature in realizability models as a way of dealing with polymorphism. Here the reason for types-as-PERs rather than types-as-sets is more basic: sets are insufficient to model the typed equational theory of the calculus.

In Section 4.6 on page 113 there is a discussion on different varieties of models for subtyping calculi, which provides further explanation for the present definition. In Chapter 4, there is an abstract model definition given for λ_{Power}, using an applicative structure and not based on a lambda model. It was developed after the model given here, and relies on the idea of *rough typing*. In Chapter 5, singleton types are included in this alternative form of model.

The definitions here vary slightly from related accounts in the literature, by incorporating type-term dependency and making other minor changes. As an unimportant matter of preference, I use the axiomatic definition of λ-model (rather than a combinatory model definition), to be explicit about the use of the characterising axioms.

Definition 3.15 (Lambda model [Hindley and Seldin, 1987]).
A *lambda model* is a triple, $\mathcal{D} = \langle D, \cdot, [\![\ ]\!]\rangle$, where D is a set, $\cdot$ is a binary operation on D and $[\![-]\!]_{\_} : \Lambda \to (Var \to D) \to D$ is an interpretation of untyped lambda-terms in an environment. If $\eta : Var \to D$, then the following axioms must hold:

VAR $\quad [\![x]\!]_\eta = \eta(x)$

APP $\quad [\![MN]\!]_\eta = [\![M]\!]_\eta \cdot [\![N]\!]_\eta$

$\alpha \quad [\![\lambda x.M]\!]_\eta = [\![\lambda y.M[y/x]]\!]_\eta$

$\xi \quad (\forall d \in D.\ [\![M]\!]_{\eta[x \mapsto d]} = [\![N]\!]_{\eta[x \mapsto d]}) \implies [\![\lambda x.M]\!]_\eta = [\![\lambda x.N]\!]_\eta$

FV $\quad (\forall x \in FV(M).\eta(x) = \eta'(x)) \implies [\![M]\!]_\eta = [\![M]\!]_{\eta'}$

$\beta \quad \forall d \in D.\ [\![\lambda x.M]\!]_\eta \cdot d = [\![M]\!]_{\eta[x \mapsto d]}$

From the above axioms (except β), we also have:

SUBST $[\![M[N/x]]\!]_\eta = [\![M]\!]_{\eta[x \mapsto [\![N]\!]_\eta]}$

An environment η' extends another η, written $\eta \subseteq \eta'$, if for all variables x:

$$\eta(x) \text{ is defined} \quad \implies \quad \eta'(x) \text{ is defined and } \eta(x) = \eta'(x)$$

We shall use products in the model, which can be defined by:

$$\begin{aligned} \langle a, b\rangle &= [\![\lambda f.fxy]\!]_{[x \mapsto a, y \mapsto b]} \\ \pi_1 p &= p \cdot [\![\lambda x.\lambda y.x]\!]_{[\,]} \\ \pi_2 p &= p \cdot [\![\lambda x.\lambda y.y]\!]_{[\,]} \end{aligned}$$

From now on, let $\mathcal{D}$ be some arbitrary fixed λ-model. The λ-model interprets untyped terms; as an informality, I omit the obvious *Erase* operation which deletes type information, so $[\![M]\!]$ abbreviates $[\![Erase(M)]\!]$. The fact that the erasure operation distributes with substitution will be used implicitly below.

Lower case letters are used to range over the domain D of $\mathcal{D}$. Partial equivalence relations on D are symmetric and transitive relations on D, i.e. subsets of $D \times D$. **PER** indicates the set of all PERs on D. The domain of R, $dom(R)$, is the set $\{\, d \mid d \ \ R \ \ d \,\}$, but we will often write $d \in R$ instead of $d \in dom(R)$. The equivalence class $\{\, d' \mid d' \ \ D \ \ d \,\}$ of d in R is written $[d]_R$ and Q(R) is the set of equivalence classes $\{\, [d]_R \mid d \in dom(R) \,\}$ in R. Inclusion of PERs, written $R \subseteq S$, is simply subset inclusion on $D \times D$.

It is well known that **PER** can be extended to a category by taking morphisms between PERs R and S to be *computable* functions between their quotients, i.e. $f : Q(R) \to Q(S)$ for which there is a $p \in D$ such that $\forall d \in dom(R) \implies f([d]_R) = [p \cdot d]_S$. This fact isn't used in what follows.

We give some constructions for building PERs; it is straightforward to check that these really do define PERs.

Definition 3.16 (PER constructions).
We define PERs to interpret the types of $\lambda_{\leq\{\}}$, as follows:

- For each atomic type κ, we assume a PER R_κ such that $\kappa \leq_{Atom} \kappa'$ implies $R_\kappa \subseteq R_{\kappa'}$.

- Let R be a PER and $S(i)$ be a PER for all $i \in dom(R)$, such that whenever $i \; R \; j$, $S(i) = S(j)$. Define the PER $\Pi(R,S)$ by:

  $$f \; \Pi(R,S) \; g \qquad \text{iff} \qquad \forall a,b.\; a \; R \; b \Rightarrow f \cdot a \; S(a) \; g \cdot b$$

  Define the PER $\Sigma(R,S)$ by:

  $$\langle a_1, b_1\rangle \; \Sigma(R,S) \; \langle a_2, b_2\rangle \qquad \text{iff} \qquad a_1 \; R \; a_2 \quad \text{and} \quad b_1 \; S(a_1) \; b_2$$

- Let R be a PER. Define the PER $[p]_R$ by:

  $$m \; [p]_R \; n \qquad \text{iff} \qquad m \; R \; n \quad \text{and} \quad m \; R \; p$$

□

Now we can give the interpretation of contexts and types. The interpretation $[\![\Gamma]\!]$ of a well-formed context Γ is a PER. The interpretation of a well-formed type in some context is a map $[\![\Gamma \rhd A]\!] : dom([\![\Gamma]\!]) \to \mathbf{PER}$ that is invariant under choice of representative of equivalence class in $[\![\Gamma]\!]$. An alternative scheme is to define $[\![\Gamma \rhd A]\!] : Q([\![\Gamma]\!]) \to \mathbf{PER}$, and then the interpretation of a term is easily made into a morphism in the category **PER**. We take the first approach here because it seems more elementary when dealing with environments. The drawback is that the interpretation is a *partial* definition, because it will be a product of the soundness theorem itself that $[\![\Gamma]\!]$ or $[\![\Gamma \rhd A]\!]_\eta$ indeed denote PERs. In any case, giving a partial definition for the interpretation function is typical of semantics for dependent type systems.

Definition 3.17 (Interpretation of contexts and types).
For each context Γ, we define a PER $[\![\Gamma]\!]$ by:

$$[\![\langle\rangle]\!] = D \times D$$

$$[\![\Gamma, x : A]\!] = \Sigma([\![\Gamma]\!], [\![\Gamma \rhd A]\!])$$

For each context Γ and type A, we define a PER $[\![\Gamma \rhd A]\!]_\eta$, for each $\eta \in dom([\![\Gamma]\!])$:

$$[\![\Gamma \rhd \kappa]\!]_\eta = R_\kappa$$

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

$$[\![\Gamma \rhd \{M\}_A]\!]_\eta = [\, [\![M]\!]_\eta{}^\Gamma \,]_{[\![\Gamma \rhd A]\!]_\eta}$$

Notice that $\triangleright$ is just used as a place-holder here, it does not signify a judgement derivation. Λ is lambda-abstraction at the meta-level, and $\eta^{\Gamma} : Var \rightharpoonup D$ is the environment defined by projections on η,

$$\begin{array}{lcl} \eta^{()}(y) & & \text{undefined, for all } y. \\ \eta^{\Gamma, x:A}(y) & = & \begin{cases} \pi_2(\eta), & \text{if } y \equiv x, \\ (\pi_1(\eta))^{\Gamma}(y) & \text{if } y \not\equiv x. \end{cases} \end{array}$$

□

This is a partial definition because the second clause for contexts is well-defined only if $[\![\Gamma \triangleright A]\!]_{\eta_1} = [\![\Gamma \triangleright A]\!]_{\eta_2}$ whenever $\eta_1 \;[\![\Gamma]\!]\; \eta_2$, and similarly for the clause for Π-types.

The following theorem establishes the main result, soundness of the interpretation. It shows the additional well-definedness property that the PER $[\![\Gamma \triangleright A]\!]_{\eta}$ is unaffected by the choice of representative $\eta \in [\![\Gamma]\!]$, whenever there is a derivation of $\Gamma \triangleright A$. The parts of the theorem need to be proven together because of the presence of dependent types.

Theorem 3.18 (Well-definedness and soundness).

1. $\triangleright \Gamma \implies [\![\Gamma]\!] \in$ *PER.*
2. $\Gamma \triangleright A$ *type* $\implies$
   $$\forall \eta_1, \eta_2.\ \eta_1 \;[\![\Gamma]\!]\; \eta_2 \implies [\![\Gamma \triangleright A]\!]_{\eta_1} = [\![\Gamma \triangleright A]\!]_{\eta_2}$$
   and $[\![\Gamma \triangleright A]\!]_{\eta_1}, [\![\Gamma \triangleright A]\!]_{\eta_2}$ *are both well-defined.*
3. $\Gamma \triangleright M : A \implies$
   $$\forall \eta_1, \eta_2.\ \eta_1 \;[\![\Gamma]\!]\; \eta_2 \implies [\![M]\!]_{\eta_1{}^{\Gamma}} \;([\![\Gamma \triangleright A]\!]_{\eta_1})\; [\![M]\!]_{\eta_2{}^{\Gamma}}$$
4. $\Gamma \triangleright A \leq B \implies$
   $$\forall \eta \in [\![\Gamma]\!].\ [\![\Gamma \triangleright A]\!]_{\eta} \subseteq [\![\Gamma \triangleright B]\!]_{\eta}$$

Equivalence classes are disjoint, so $[m]_R \subseteq [n]_R$ implies $m \;R\; n$ if $[m]_R$ is non-empty. The consequence of part 2 of the theorem implies that the interpretation of $\{M\}_A$ is non-empty, thus $\Gamma \triangleright \{M\}_A \leq \{N\}_A$ implies that M and N are equal at type A in the PER model.

To prove Theorem 3.18 we require some auxiliary propositions establishing properties of the definitions.

Lemma 3.19 (Semantic weakening).
Let Γ_1, Γ_2 *be two contexts with* $\eta_1 \in [\![\Gamma_1]\!]$, $\eta_2 \in [\![\Gamma_2]\!]$, *and* $\eta_1{}^{\Gamma_1} \subseteq \eta_2{}^{\Gamma_2}$.
Then:

1. *For terms* M *with* $FV(M) \subseteq dom(\Gamma_1)$, $\quad [\![M]\!]_{\eta_1{}^{\Gamma_1}} = [\![M]\!]_{\eta_2{}^{\Gamma_2}}$

2. *For types A with $FV(A) \subseteq dom(\Gamma_1)$,* $[\![\Gamma_1 \rhd A]\!]_{\eta_1} = [\![\Gamma_2 \rhd A]\!]_{\eta_2}$

Proof The first part is immediate by the assumptions and axiom (FV) of $\mathcal{D}$. We show the second holds for all Γ_1, η_1, Γ_2, η_2 by induction on A.

Case $A \equiv \kappa$: Trivial.

Case $A \equiv \Pi x : B.C$:
By the induction hypothesis, $[\![\Gamma_1 \rhd B]\!]_{\eta_1} = [\![\Gamma_2 \rhd B]\!]_{\eta_2}$.
Let $a \in [\![\Gamma_1 \rhd B]\!]_{\eta_1}$. Then $[\![\Gamma_1, x : A \rhd B]\!]_{\langle \eta_1, a\rangle} = [\![\Gamma_2, x : A \rhd B]\!]_{\langle \eta_2, a\rangle}$ by the induction hypothesis, since $\langle \eta_1, a\rangle \in [\![\Gamma_1, x : A]\!]$ and $\langle \eta_2, a\rangle \in [\![\Gamma_2, x : A]\!]$.
Hence $[\![\Gamma_1 \rhd \Pi x : B.C]\!]_{\eta_1} = [\![\Gamma_2 \rhd \Pi x : B.C]\!]_{\eta_2}$.

Case $A \equiv \{M\}_B$:
By the induction hypothesis, $[\![\Gamma_1 \rhd B]\!]_{\eta_1} = [\![\Gamma_2 \rhd B]\!]_{\eta_2}$.
By part 1, $[\![M]\!]_{\eta_1}{}^{\Gamma_1} = [\![M]\!]_{\eta_2}{}^{\Gamma_2}$.
Hence $[\![\Gamma_1 \rhd \{M\}_B]\!]_{\eta_1} = [\![\Gamma_2 \rhd \{M\}_B]\!]_{\eta_2}$. □

Lemma 3.20 (Semantic contexts).
$\forall \eta_1, \eta_2. \forall x \in dom(\Gamma).\ \eta_1\ [\![\Gamma]\!]\ \eta_2 \Longrightarrow \eta_1{}^{\Gamma}(x)\ [\![\Gamma \rhd \Gamma(x)]\!]_{\eta_1}\ \eta_2{}^{\Gamma}(x)$.

Proof For all $\Gamma = \Gamma_1$ by induction on the structure of Γ_1:

Case $\Gamma_1 \equiv \langle\rangle$: vacuous.

Case $\Gamma_1 \equiv \Gamma, y : A$: Let $\langle \eta_1, a_1\rangle\ [\![\Gamma, y : A]\!]\ \langle \eta_2, a_2\rangle$. Then:

$$\begin{array}{c} \eta_1\ [\![\Gamma]\!]\ \eta_2 \\ a_1\ [\![\Gamma \rhd A]\!]_{\eta_1}\ a_2 \qquad (**) \end{array}$$

by the definition of $\Sigma(R, S)$. Assume $x \in dom(\Gamma_1)$.
If $x \equiv y$ then $\langle \eta_1, a_1\rangle^{\Gamma, y:A}(y) = a_1$ and $\langle \eta_2, a_2\rangle^{\Gamma, y:A}(y) = a_2$, so the result by (**) and Lemma 3.19.
If $x \not\equiv y$ then by the induction hypothesis,

$$\begin{array}{ll} & \eta_1{}^{\Gamma}(x)\ [\![\Gamma \rhd \Gamma(x)]\!]_{\eta_1}\ \eta_2{}^{\Gamma}(x) \\ \Rightarrow & \langle \eta_1, a_1\rangle^{\Gamma, y:A}(x)\ [\![\Gamma, y : A \rhd \Gamma(x)]\!]_{\langle \eta_1, a_1\rangle}\ \langle \eta_1, a_1\rangle^{\Gamma, y:A}(x) \end{array}$$

by the definition of η^{Γ} and by Lemma 3.19. □

Lemma 3.21 (Relating semantic and syntactic substitution).
Let $\Gamma, x : A, \Gamma'$ be a context, $n = [\![N]\!]_{\eta^\Gamma}$ for some term N, some $\eta \in [\![\Gamma]\!]$. For $\eta_1 \in [\![\Gamma', \Gamma[N/x]]\!]$, $\eta_2 \in [\![\Gamma, x : A, \Gamma']\!]$ with $\eta^\Gamma \subseteq {\eta_1}^{\Gamma', \Gamma[N/x]} \subseteq {\eta_2}^{\Gamma, x:A, \Gamma'}$ where ${\eta_2}^{\Gamma, x:A, \Gamma'}(x) = n$, we have:

1. *For terms M with $FV(M) \subseteq dom(\Gamma, x : A, \Gamma')$,*

$$[\![M[N/x]]\!]_{{\eta_1}^{\Gamma', \Gamma[N/x]}} = [\![M]\!]_{{\eta_2}^{\Gamma, x:A, \Gamma'}}$$

2. *For types B with $FV(B) \subseteq dom(\Gamma, x : A, \Gamma')$,*

$$[\![\Gamma', \Gamma[N/x] \rhd B[N/x]]\!]_{\eta_1} = [\![\Gamma, x : A, \Gamma' \rhd B]\!]_{\eta_2}$$

Proof The first part follows from axioms (SUB),(FV) in $\mathcal{D}$ and semantic weakening, Lemma 3.19 above. For the second part, use induction on B:

Case $B \equiv \kappa$: Trivial.

Case $B \equiv \Pi y : C.D$:
By the induction hypothesis, $[\![\Gamma', \Gamma[N/x] \rhd C[N/x]]\!]_{\eta_1} = [\![\Gamma, x : A, \Gamma' \rhd C]\!]_{\eta_2}$.
Let $c \in [\![\Gamma', \Gamma[N/x] \rhd C[N/x]]\!]_{\eta_1}$. Then by the induction hypothesis,
$[\![\Gamma', \Gamma[N/x], y : C[N/x] \rhd D[N/x]]\!]_{\langle \eta_1, c\rangle} = [\![\Gamma, x : A, \Gamma', y : C \rhd D]\!]_{\langle \eta_2, c\rangle}$
The result is seen by the definition of $\Pi(R, S)$.

Case $B \equiv \{M\}_C$: By the induction hypothesis, we have the result for C. The result follows by part 1 for terms and the definition of interpretation for $\{M\}_C$. □

Now we can prove the main result.

Proof of Theorem 3.18 The parts are proved simultaneously by induction on derivations. The proof ends on page 76.

Part 1. Easy, using part 2.

Part 2. We use a nested induction on the structure of A, which circumvents the lack of well-formedness premises in (FORM-Π) and (FORM-{}), see the remarks in Section 3.4.

Case (FORM-ATOMIC): Trivial.

Case (FORM-Π): By IH, we have

$$[\![\Gamma \rhd A]\!]_{\eta_1} = [\![\Gamma \rhd A]\!]_{\eta_2}$$
$$\forall \eta_3, \eta_4.\ \eta_3\ [\![\Gamma, x : A]\!]\ \eta_4 \implies [\![\Gamma, x : C \rhd B]\!]_{\eta_3} = [\![\Gamma, x : C \rhd B]\!]_{\eta_4}$$

So $\forall a \in [\![\Gamma \rhd A]\!]_{\eta_1}.[\![\Gamma, x : A \rhd B]\!]_{\langle \eta_1, a\rangle} = [\![\Gamma, x : A \rhd B]\!]_{\langle \eta_2, a\rangle}$.
Hence $[\![\Gamma \rhd \Pi x{:}A.B]\!]_{\eta_1} = [\![\Gamma \rhd \Pi x{:}A.B]\!]_{\eta_2}$ by the definition of $\Pi(R, S)$.

Case (FORM-{}): By the IH we have that $[\![\Gamma \rhd A]\!]_{\eta_1} = [\![\Gamma \rhd A]\!]_{\eta_2}$.
We use induction on the derivation here; by the IH for part 2:

$$[\![M]\!]_{\eta_1}{}^{\Gamma} \in [\![\Gamma \rhd A]\!]_{\eta_1}$$
$$[\![M]\!]_{\eta_2}{}^{\Gamma} \in [\![\Gamma \rhd A]\!]_{\eta_2} = [\![\Gamma \rhd A]\!]_{\eta_1}$$

Hence

$$[\![M]\!]_{\eta_1}{}^{\Gamma} \;\; [\![\Gamma \rhd A]\!]_{\eta_1} \;\; [\![M]\!]_{\eta_2}{}^{\Gamma}.$$

Part 3.

Case (VAR): $[\![x]\!]_{\eta_1}{}^{\Gamma} = \eta_1{}^{\Gamma}(x)$ and similarly for η_2; the result follows by Lemma 3.20.

Case (λ): Let $f_1 = [\![\lambda x.M]\!]_{\eta_1}{}^{\Gamma}$ and f_2 similarly. We wish to show

$$f_1 \;\; [\![\Gamma \rhd \Pi x{:}A.B]\!]_{\eta_1} \;\; f_2$$

which holds iff for all a_1, a_2:

$$a_1 \;\; [\![\Gamma \rhd A]\!]_{\eta_1} \;\; a_2 \Longrightarrow f_1 \cdot a_1 \;\; [\![\Gamma, x : A \rhd B]\!]_{\langle \eta_1, a_1 \rangle} \;\; f_2 \cdot a_2$$

Suppose the antecedent. By axiom (β),

$$f_1 \cdot a_1 = [\![M]\!]_{\eta_1}{}^{\Gamma}[x \mapsto a_1]$$

where

$$\eta_1{}^{\Gamma}[x \mapsto a_1] \;=\; \langle \eta_1, a_1 \rangle^{\Gamma, x:A} \in [\![\Gamma, x : A]\!]$$

and similarly for f_2, η_2, a_2. So we can apply the IH for the premise to get

$$[\![M]\!]_{\langle \eta_1, a_1 \rangle^{\Gamma, x:A}} \;\; [\![\Gamma, x : A \rhd B]\!]_{\langle \eta_1, a_1 \rangle} \;\; [\![M]\!]_{\langle \eta_2, a_2 \rangle^{\Gamma, x:A}}$$

which establishes the result.

Case (APP): Let $m_1 = [\![M]\!]_{\eta_1}{}^{\Gamma}$, and m_2, n_1, n_2 similarly.
By IH we have

$$m_1 \;\; [\![\Gamma \rhd \Pi x{:}A.B]\!]_{\eta_1} \;\; m_2$$
$$n_1 \;\; [\![\Gamma \rhd A]\!]_{\eta_1} \;\; n_2$$

So by the definition of $\Pi(R, S)$,

$$m_1 \cdot n_1 \;\; [\![\Gamma, x : A \rhd B]\!]_{\langle \eta_1, n_1 \rangle} \;\; m_2 \cdot n_2$$

By Lemma 3.21, $[\![\Gamma, x : A \rhd B]\!]_{\langle \eta_1, n_1 \rangle} = [\![\Gamma \rhd B[N/x]]\!]_{\eta_1}$, so

$$m_1 \cdot n_1 \;\; [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \;\; m_2 \cdot n_2$$

Finally by the (APP) axiom for the model,

$$[\![MN]\!]_{\eta_1}{}^{\Gamma} \;\; [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \;\; [\![MN]\!]_{\eta_2}{}^{\Gamma}.$$

Case (SUB): By IH for the first premise,

$$[\![M]\!]_{\eta_1} \ [\![\Gamma \rhd A]\!]_{\eta_1} \ [\![M]\!]_{\eta_2}$$

and by IH for part 3 and the second premise,

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

hence the result

$$[\![M]\!]_{\eta_1} \ [\![\Gamma \rhd B]\!]_{\eta_1} \ [\![M]\!]_{\eta_2}.$$

Case (EQ-REFL): By IH,

$$[\![M]\!]_{\eta_1} \ [\![\Gamma \rhd A]\!]_{\eta_1} \ [\![M]\!]_{\eta_2}$$

and so also

$$[\![M]\!]_{\eta_1} \ [\![\Gamma \rhd A]\!]_{\eta_1} \ [\![M]\!]_{\eta_1}$$

hence

$$[\![M]\!]_{\eta_1} \ [\![\Gamma \rhd \{M\}_A]\!]_{\eta_1} \ [\![M]\!]_{\eta_2}.$$

Case (EQ-λ): Let $f_1 = [\![\lambda x.M]\!]_{\eta_1}{}^{\Gamma}$, $f_1' = [\![\lambda x.M']\!]_{\eta_1}{}^{\Gamma}$, $f_2 = [\![\lambda x.M]\!]_{\eta_1}{}^{\Gamma}$. We wish to show

$$\begin{array}{l} f_1 \ [\![\Gamma \rhd \Pi x{:}A'.B']\!]_{\eta_1} \ f_2 \\ f_1 \ [\![\Gamma \rhd \Pi x{:}A'.B']\!]_{\eta_1} \ f_1' \end{array}$$

which holds iff for all a_1, a_2 :

$$a_1 \ [\![\Gamma \rhd A]\!]_{\eta_1} \ a_2 \quad \Rightarrow \quad \begin{array}{l} f_1 \cdot a_1 \ [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta_1, a_1 \rangle} \ f_2 \cdot a_2 \\ f_1 \cdot a_1 \ [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta_1, a_1 \rangle} \ f_1' \cdot a_2 \end{array}$$

Suppose the antecedent. By axiom (β),

$$\begin{array}{l} f_1 \cdot a_1 = [\![M]\!]_{\eta_1}{}^{\Gamma}[x \mapsto a_1] \\ f_2 \cdot a_2 = [\![M]\!]_{\eta_2}{}^{\Gamma}[x \mapsto a_2] \\ f_1' \cdot a_1 = [\![M']\!]_{\eta_1}{}^{\Gamma}[x \mapsto a_1] \end{array}$$

And as before, we can apply the IH for the 2nd premise to get

$$\begin{array}{l} [\![M]\!]_{\langle \eta_1, a_1 \rangle}{}^{\Gamma, x:A'} \ [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta_1, a_1 \rangle} \ [\![M]\!]_{\langle \eta_2, a_2 \rangle}{}^{\Gamma, x:A'} \\ [\![M]\!]_{\langle \eta_1, a_1 \rangle}{}^{\Gamma, x:A'} \ [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta_1, a_1 \rangle} \ [\![M']\!]_{\langle \eta_1, a_1 \rangle}{}^{\Gamma, x:A'} \end{array}$$

which establishes the result.

Case (EQ-APP): Let $m_1 = [\![M]\!]_{\eta_1} r$, and $m'_1, m_2, n_1, n'_1, n_2$ similarly.
By IH we have

$$\begin{aligned} m_1 \ [\![\Gamma \rhd \Pi x{:}A.\,B]\!]_{\eta_1} \ m_2 \\ m_1 \ [\![\Gamma \rhd \Pi x{:}A.\,B]\!]_{\eta_1} \ m'_1 \\ n_1 \ [\![\Gamma \rhd A]\!]_{\eta_1} \ n_2 \\ n_1 \ [\![\Gamma \rhd A]\!]_{\eta_1} \ n'_1 \end{aligned}$$

So, as for (APP),

$$\begin{aligned} m_1 \cdot n_1 \ [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \ m_2 \cdot n_2 \\ m_1 \cdot n_1 \ [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \ m'_1 \cdot n'_1 \end{aligned}$$

and

$$\begin{aligned} [\![MN]\!]_{\eta_1} r \ [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \ [\![MN]\!]_{\eta_2} r \\ [\![MN]\!]_{\eta_1} r \ [\![\Gamma \rhd B[N/x]]\!]_{\eta_1} \ [\![M'N']\!]_{\eta_1} r \end{aligned}$$

which establishes the case.

Part 4.

Case (SUB-REFL): Trivial.

Case (SUB-TRANS): By the induction hypothesis and transitivity of $\subseteq$.

Case (SUB-ATOMIC): By the restriction that $R_K \subseteq R_{K'}$.

Case (SUB-Π): We wish to show

$$[\![\Gamma \rhd \Pi x{:}A.\,B]\!]_\eta \subseteq [\![\Gamma \rhd \Pi x{:}A'.\,B']\!]_\eta$$

Let $f \in [\![\Gamma \rhd \Pi x{:}A.\,B]\!]_\eta$. So for all a_1, a_2,

$$a_1 \ [\![\Gamma \rhd A]\!]_\eta \ a_2 \quad \Longrightarrow \quad f \cdot a_1 \ [\![\Gamma, x : A \rhd B]\!]_{\langle \eta, a_1 \rangle} \ f \cdot a_2$$

By the induction hypothesis for the first premise, $[\![\Gamma \rhd A']\!]_\eta \subseteq [\![\Gamma \rhd A]\!]_\eta$. Observe that in the definition of $[\![\Gamma \rhd -]\!]_\eta$, types play no role except to ensure that variables in the environment g inhabit their claimed types, in Lemma 3.20. So we may replace $[\![\Gamma, x : A \rhd B]\!]$ by $[\![\Gamma, x : A' \rhd B]\!]$,

$$a_1 \ [\![\Gamma \rhd A']\!]_\eta \ a_2 \quad \Longrightarrow \quad f \cdot a_1 \ [\![\Gamma, x : A' \rhd B]\!]_{\langle \eta, a_1 \rangle} \ f \cdot a_2$$

By the induction hypothesis for the second premise,

$$[\![\Gamma, x : A' \rhd B]\!]_{\langle \eta, a_1 \rangle} \subseteq [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta, a_1 \rangle}$$

hence

$$a_1 \ [\![\Gamma \rhd A']\!]_\eta \ a_2 \quad \Longrightarrow \quad f \cdot a_1 \ [\![\Gamma, x : A' \rhd B']\!]_{\langle \eta, a_1 \rangle} \ f \cdot a_2$$

which implies $f \in [\![\Gamma \rhd \Pi x{:}A'.\,B']\!]_\eta$ as required.

Case (SUB-{}): By the definition of $[m]_R$.

Case (SUB-EQ-SYM): We wish to show

$$[\![\Gamma \rhd \{N\}_A]\!]_\eta \subseteq [\![\Gamma \rhd \{M\}_B]\!]_\eta$$

Let $n \in [\![\Gamma \rhd \{N\}_A]\!]_\eta$, so

$$n \ [\![\Gamma \rhd A]\!]_\eta \ [\![N]\!]_{\eta^\Gamma}$$

By IH for part 2 and the first premise,

$$[\![M]\!]_{\eta^\Gamma} \ [\![\Gamma \rhd A]\!]_\eta \ [\![N]\!]_{\eta^\Gamma}$$

So

$$n \ [\![\Gamma \rhd A]\!]_\eta \ [\![M]\!]_{\eta^\Gamma}$$

By IH for the second premise, $[\![\Gamma \rhd A]\!]_\eta \subseteq [\![\Gamma \rhd B]\!]_\eta$, hence

$$n \ [\![\Gamma \rhd B]\!]_\eta \ [\![M]\!]_{\eta^\Gamma}$$

and so

$$n \in [\![\Gamma \rhd \{M\}_B]\!]_\eta$$

which establishes the result.

Case (SUB-EQ-ITER): This follows from the definition of $[m]_A$. Since

$$a \ [m]_A \ b \quad \text{iff} \quad (a \ A \ b) \ \bigwedge \ (a \ A \ m)$$

whereas

$$\begin{array}{lll} a \ [m]_{[m]_A} \ b & \text{iff} & (a \ [m]_A \ b) \ \bigwedge \ (a \ [m]_A \ m) \\ & \text{iff} & (a \ A \ b) \ \bigwedge \ (a \ A \ m) \ \bigwedge \ (a \ A \ m) \\ & & \bigwedge \ (a \ A \ m) \\ & \text{iff} & (a \ A \ b) \ \bigwedge \ (a \ A \ m). \end{array}$$

□

3.7 Discussion

This section discusses alternative formulations of $\lambda_{\leq\{\}}$ and related work. Section 3.8 concludes the chapter.

3.7.1 *Alternative formulations*

An alternative presentation of the system can be given which has no typing judgement, but recovers typing by subtyping of singleton types, thus:

$$\Gamma \rhd M : A \qquad \Longleftrightarrow \qquad \exists B.\, \Gamma \rhd \{M\}_B \leq A$$

This replacement works more smoothly with *untagged* singletons, where

$$\Gamma \rhd M : A \qquad \Longleftrightarrow \qquad \Gamma \rhd \{M\} \leq A$$

The system with untagged singletons has some interesting admissible rules — for example, the rule (EQ-λ) is admissible in the presence of the weaker (EQ-λ-EQUAL-BOUNDS) rule shown on page 54, via (SUB-Π). Without a tag, $\{M\}$ lifts some subtype polymorphism to the type level. However, the system with untagged singletons does not interpret a typed equational theory, or relate clearly to a PER model.

A "wrapped-up" version of $\lambda_{\leq\{\}}$ with only a subtyping judgement form suggests new directions. One is the possibility of unifying terms and types by identifying a term M with its singleton type $\{M\}$. There are numerous disparate cases where researchers have found cause to replace or augment a typing relation with a pre-order over terms in this way [Mosses, 1989, Feijs, 1989, Levy et al., 1991, Dami, 1995, for example]. In a wrapped-up $\lambda_{\leq\{\}}$ we still have a good distinction between types and terms because the latter are always warmly insulated in singleton braces. The PER model reflects this distinction. Although subtyping imposes an ordering on terms here, it degenerates to equality within the same type; another direction to study would be a variant of $\lambda_{\leq\{\}}$ where the ordering on terms signifies some kind of refinement principle rather than equality. This would integrate nicely into the model of program development, so that we have a refinement sequence of the form $SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \rightsquigarrow \{P_1\} \rightsquigarrow \cdots \{P_n\}$. The model definition given in the next chapter could be extended to allow this kind of interpretation (see Section 4.6.1).

The presentation can be abbreviated still further. For example, the context judgement can be removed in preference for a weakening rule, in the style of Pure Type Systems [Barendregt, 1992]. This would give a theory with just two judgement forms (and doing this for the theory in the next chapter would leave just one judgement form). While removing the context judgement makes for a concise presentation and some slightly shorter syntactic proofs, it seems to have little other benefit.

3.7.2 Related work

During this work, I drew inspiration from research into type systems for object-oriented programming languages. The system most studied is the extension of System F with subtyping called $F_{\leq}$. Variants of $F_{\leq}$ and its PER semantics are described by Bruce and Longo [1990], Scedrov [1990], and Cardelli and Longo [1991]. The equational theory of $F_{\leq}$ is investigated in Cardelli et al. [1992]. So far none of these systems has dependent types; there are examples of PER semantics for dependent type systems in the literature, but not for dependent subtyping systems. Here I have shown that the extension is smooth, at least for a containment semantics. The next chapter extends this to a calculus with type-valued functions and type-term abstraction.

ATT and ATTT Susumu Hayashi's work [1994] was mentioned in Section 3.1. He describes two systems with singleton, union, and intersection types. The first system is called ATT. Apart from the presence of more type constructors, ATT differs from $\lambda_{\leq\{\}}$ in these aspects:

- ATT is based on untyped λ-calculus, so λ-abstractions are untyped;
- ATT has primitive rules for type conversion and subject reduction (he added subject reduction because he couldn't prove it — counterexamples are known for similar systems);
- Function spaces are restricted to be non-dependent, i.e., of the form $A \to B$, but dependent products can be encoded using the other type constructors;
- There is no subtyping judgement or equality judgement, but at least the second of these can also be encoded;
- The rules for singleton elimination differ, see below.

Hayashi gives a set-theoretic semantics for ATT, and then goes on to describe his second system, ATTT. This system is typed; it extends System F with the same type constructors as before, but this time the "non-informative" types are treated specially, as *refinements* of the usual F types. This achieves a weak separation between typing and specification, so that type-checking remains decidable although refinement checking is not. I develop a similar scheme for ASL+, beginning from the *rough typing* system introduced in Section 4.5; see also the discussion in Section 4.9.1 about related work by Pfenning [1993].

My rules for singleton types differ from Hayashi's. He has two rules for singleton elimination. The simpler one

$$\frac{\Gamma \rhd M : \{N\}_A}{\Gamma \rhd M : A}$$

is admissible in $\lambda_{\leq\{\}}$ by Corollary 3.6. The other rule expresses a replacement scheme, and reflects Hayashi's intention to encode a constructive logic. This strong rule treats the intersection of two singletons as a propositional equality:

$$\frac{\Gamma \rhd P : \{M\}_A \cap \{N\}_A \qquad \Gamma \rhd Q : B[M/x] \qquad \begin{array}{l}\Gamma \rhd M : A \\ \Gamma \rhd N : A \\ \Gamma, x : A \rhd B\end{array}}{\Gamma \rhd Q : B[N/x]}$$

Hayashi expresses doubt that this elimination rule is the only "right" one.

Hayashi's systems are powerful and offer a new perspective on type theory for program extraction: constructive, but slightly less so. The motivations here are less grandiose, but it would be interesting to compare extensions of $\lambda_{\leq\{\}}$ with ATT and ATTT more formally.

3.8 Summary

This chapter presented the calculus $\lambda_{\leq\{\}}$, which extends $\lambda_{\leq}$, the simply typed lambda calculus with subtyping. The new feature is singleton types, which also introduce type-term dependency into the system. The syntax was described in Section 3.3 and is summarised in Figures 3.1–3.4, starting on page 80. A PER model was given for $\lambda_{\leq\{\}}$ and proven sound, in Section 3.6. In the model, singleton types $\{M\}_A$ are interpreted as the equivalence class of $[\![M]\!]$ in the PER $[\![A]\!]$. This is the first PER model for a system with subtyping and dependent types.

The aim was to establish some basic meta-theory of subtyping in the dependent setting, without the complication of functions from terms to types or (bounded) polymorphism that would be in a more realistic system. A richer system is an obvious next step, which is taken in the next chapter when I introduce a system with power types.

The important results in this chapter are the basic properties of the presentation of $\lambda_{\leq\{\}}$ established in Section 3.4, which show that it is a sensible system; the further properties shown in Section 3.5, which are the minimal type property, Theorem 3.14, and subject reduction, Theorem 3.11; and the soundness result for the PER model definition, Theorem 3.18.

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

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

$$\frac{\rhd \Gamma}{\Gamma \rhd P \ \mathit{type}} \qquad \text{(FORM-ATOMIC)}$$

$$\frac{\Gamma, x : A \rhd B \ \mathit{type}}{\Gamma \rhd \Pi x{:}A.\,B} \qquad \text{(FORM-}\Pi\text{)}$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A \ \mathit{type}} \qquad \text{(FORM-}\{\}\text{)}$$

Figure 3.1: Context and type formation for $\lambda_{\leq\{\}}$

$$\frac{\rhd \Gamma}{\Gamma \rhd x : \Gamma(x)} \qquad \text{(VAR)}$$

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

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

$$\frac{\Gamma \rhd M : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd M : B} \qquad \text{(SUB)}$$

Figure 3.2: Typing for $\lambda_{\leq\{\}}$

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

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

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

Note: $\Gamma \rhd M = N : A$ is short for $\Gamma \rhd M : \{N\}_A$.

Figure 3.3: Equality for $\lambda_{\leq\{\}}$

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

$$\frac{\Gamma \rhd A \leq B \qquad \Gamma \rhd B \leq C}{\Gamma \rhd A \leq C} \qquad \text{(SUB-TRANS)}$$

$$\frac{\rhd \Gamma \qquad \kappa \leq_{Atom} \kappa'}{\Gamma \rhd \kappa \leq \kappa'} \qquad \text{(SUB-ATOMIC)}$$

$$\frac{\Gamma \rhd A' \leq A \qquad \Gamma, x : A' \rhd B \leq B' \qquad \Gamma, x : A \rhd B}{\Gamma \rhd \Pi x{:}A.\, B \leq \Pi x{:}A'.\, B'} \qquad \text{(SUB-}\Pi\text{)}$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A \leq A} \qquad \text{(SUB-\{\})}$$

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd \{N\}_A \leq \{M\}_B} \qquad \text{(SUB-EQ-SYM)}$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A \leq \{M\}_{\{M\}_A}} \qquad \text{(SUB-EQ-ITER)}$$

Figure 3.4: Subtyping for $\lambda_{\leq\{\}}$

$$\frac{\begin{array}{l}\Gamma \rhd A' \leq A \\ \Gamma, x : A' \rhd M : B' \qquad \Gamma, x : A \rhd M : B\end{array}}{\Gamma \rhd \lambda x{:}A.\, M : \Pi x{:}A'.\, B'} \quad (\lambda\text{-NARROW})$$

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

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

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd M = N : B} \quad (\text{EQ-SUB})$$

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

$$\frac{\Gamma \rhd M = N : A \qquad \Gamma \rhd A \leq B}{\Gamma \rhd \{M\}_A \leq \{N\}_B} \quad (\text{SUB-EQ})$$

Figure 3.5: Admissible rules of $\lambda_{\leq\{\}}$

4 Power Types

A typed λ-calculus called λ_{Power} is introduced, which is a predicative fragment 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 system of dependent types.

This chapter contains the first in-depth study of power types. Basic properties of λ_{Power} are proved, and it is given a model definition in the style 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 structure the model definition, and prove strong normalization of the calculus.

4.1 Subtyping via Power Types

POWER TYPES were introduced by Cardelli [1988], who explained the idea that *Power*(A) is the type "whose elements are all of the subtypes of the type A,"

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

Instead of a separate definition of subtyping, a relation between types is induced by inhabitation of power types:

$$A \leq B \quad =_{\mathrm{def}} \ A : \mathit{Power}(B)$$

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

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

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

The first rule makes the induced subtyping relation reflexive. The second rule is the characteristic rule of subtyping called *subsumption*, which adds subtype polymorphism to the system. Together with the third rule, this makes the induced subtyping relation transitive. Other rules capture the subtyping behaviour of type constructors.

The main motivation Cardelli had for power types was to encode *bounded* type abstraction and quantification by the usual λ-abstraction and dependent function space,

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

This simplifies the type system, since there is no need to add new constructs to the language.

Cardelli's 1988 system was meant as a flexible type system for programming languages, particularly languages with object-oriented features. Bounded type abstraction and quantification feature 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 $\Lambda\alpha{:}\mathit{Type}.M$ 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. (Further explanation of the approach and examples can be found in Gunter and Mitchell [1994].) My application of bounded abstraction is different: subtyping approximates specification refinement, and we write $\Lambda X \leq SP.M$ for the function which can be applied to any specification refining SP.

As well as power types, Cardelli's 1988 system has universal polymorphism via the *Type* : *Type* rule, dependent sum and product types, recursive types, variant types, record types and abstract types. The system is so powerful, in fact, that it is inconsistent when viewed as a logic — every type is inhabited in the empty context — and type-checking is undecidable. This was no surprise; Cardelli presented the system as a showcase combining most of the typing ideas that he and others had studied in the preceding decade.

Since 1988, Cardelli and others have studied various fragments of the full system, notably in the type system of Quest [Cardelli and Longo, 1991, Bruce and Longo, 1990]. Quest uses power *kinds* to capture subtyping, so that $\mathit{Power}(A)$ 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.[1]

Power types seem not to have been studied since. This chapter contains part of the original contribution of the thesis: the first in-depth study of power types. A *predicative* fragment of Cardelli's system is defined, called λ_{Power}. It forms the core of the type system underlying ASL+. The formal definition of λ_{Power} is given in Section 4.2; examples of its expressibility follow in Section 4.3. The remaining sections in the chapter explore basic meta-theory of the system and a semantics for it. In particular, the semantics and some of the meta-theory is based on a system for *rough typing*, which assigns "rough" non-dependent types to λ_{Power} terms. Rough typing is introduced in Section 4.5. Some discussion and comparison with related work appears at the end in Section 4.9 and Section 4.10 concludes with a summary.

4.2 The System λ_{Power}

First we define the context-free abstract syntax. Let $\mathcal{K}$ be a set of atomic type constants. The set $\boldsymbol{T}_{\mathcal{K}}$ of λ_{Power} *pre-terms over* $\mathcal{K}$ is given by the grammar:

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

(writing $\boldsymbol{T}$ for short), where $\boldsymbol{V}$ is a countable infinite set of variables. I 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 the 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

[1]Luca Cardelli told me this.

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 $\Pi x{:}A.\,\Pi y{:}A.\,B$.

Not all pre-terms make sense. The well-formed pre-terms consist of *terms* and *types*, defined in Definition 4.1 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.

A *pre-context* is a sequence of variable declarations $x_1 : A_1, x_2 : A_2 \ldots$ with the stipulation that no variable is declared more than once. The empty pre-context is sometimes written as $\langle\rangle$ to draw attention to its presence, otherwise it is invisible. The meta-variables $\Gamma, \Delta, \ldots$ range over pre-contexts.

4.2.1 Presentation of λ_{Power}

The system λ_{Power} is defined using three forms of judgement:

$\rhd\ \Gamma$	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

The judgements are defined simultaneously by the rules in Figures 4.1 and 4.2. As usual, I shall write $\Gamma \rhd M : A$ to mean "the judgement $\Gamma \rhd M : A$ is derivable," etc.

Here is a brief outline of the rules. Many rules are similar to those in $\lambda_{\leq\{\}}$, which were explained in detail in Section 3.3.1 on page 51.

Context formation (Figure 4.2).
These rules are standard. The rule (EMPTY) says that the empty context is valid and (EXTEND) adds declarations to contexts. The judgement $\Gamma \rhd A : \mathit{Power}(B)$ 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 4.1).
The rules (VAR), (λ), (APP) and (SUB) are standard. The rule (CONV) is also standard, and ensures that equal types have the same elements. The rule (ATOMIC)

$$\frac{\rhd \Gamma}{\Gamma \rhd \kappa : \mathit{Power}(\kappa)} \qquad \text{(ATOMIC)}$$

$$\frac{\rhd \Gamma \qquad x \in \mathit{Dom}(\Gamma)}{\Gamma \rhd x : \Gamma(x)} \qquad \text{(VAR)}$$

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

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

$$\frac{\Gamma \rhd M : A \qquad \Gamma \rhd A : \mathit{Power}(B)}{\Gamma \rhd M : B} \qquad \text{(SUB)}$$

$$\frac{\Gamma \rhd M : \Pi \vec{x}{:}\vec{A}.\ \mathit{Power}(B)}{\Gamma \rhd M : \Pi \vec{x}{:}\vec{A}.\ \mathit{Power}(M\,\vec{x})} \qquad \text{(REFL)}$$

$$\frac{\Gamma \rhd M : A \qquad \Gamma \rhd A = B : \mathit{Power}(C)}{\Gamma \rhd M : B} \qquad \text{(CONV)}$$

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

$$\frac{\Gamma \rhd A : \mathit{Power}(B)}{\Gamma \rhd \mathit{Power}(A) : \mathit{Power}(\mathit{Power}(B))} \qquad (\mathit{Power})$$

Figure 4.1: Typing for λ_{Power}

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

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

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

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

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

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

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

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

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

Figure 4.2: Context formation and equality for λ_{Power}

introduces atomic types into the system; each atomic type is a subtype of itself, and so is self-evidently well-formed.

The rule (REFL) is a rule-scheme; the vector notation is shorthand for:

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

We can define a derived subtyping relation for higher types by extending the subtype relation pointwise: if a function M has a type $\Pi x{:}A.\ \mathit{Power}(B)$, this expresses that it is a subtype of B pointwise (this is explained further in Section 4.3.2). For each $n \geq 0$, the rule (REFL) asserts reflexivity of the induced subtype relation for n-ary type-valued functions.[2] Reflexivity of subtyping for types is the case that $n = 0$.

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

The rule (*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 4.2).
These rules are standard. In the rule (EQ-η), the usual side condition $x \notin FV(M)$ is intended. With subtyping, the rules have some more general consequences, shown in Proposition 4.9.

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

Definition 4.1 (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, $\Gamma \rhd A : \mathit{Power}(B)$.
3. A is a *subtype* of B in Γ if $\Gamma \rhd A : \mathit{Power}(B)$.

Sometimes I shall use the adjective "well-formed" to emphasise that a term or type can be typed in the calculus, as required by Definition 4.1. (This is to avoid confusion when another kind of type enters the picture in Section 4.5.) And I will use these derived judgement forms:

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

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

Section 4.4 shows that these definitions make sense, by proving that (amongst other things), the subtyping and type equality relations are each reflexive and transitive.

In this chapter 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 4.1); this is to be sure that the presence of power types is not forgotten!

4.3 Examples in λ_{Power}

As a calculus of functions, λ_{Power} is no more expressive than the simply-typed λ-calculus.[3] In contrast with Cardelli's system, it is predicative. It is not possible to write a function which operates on any type, so there is no universal polymorphism in the style of System F. Instead we can abstract over subtypes of types, or subtypes of subtypes, and so on.

Despite this it is still possible to express complex typing ideas, because of the combination of subtyping, dependent types and 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 Aspinall and Compagnoni [1996]. I shall introduce a few examples in this section, hoping that they will help you both to understand the system and to persuade you that interest in λ_{Power} may reach beyond ASL+.

The examples are in three sub-sections: programming-language style examples in Section 4.3.1, the use of power types for higher-order subtyping in Section 4.3.2, and the use of λ_{Power} as a logical framework, in Section 4.3.3. The main application, to ASL+, is studied in other chapters. A prototype type-checker was used to develop and check the examples in this section.

4.3.1 A programming example

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

[3] If a term 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 4.5.

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

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

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

We can write a function to apply the inverse of a permutation of $\{\, 1, \ldots, n \,\}$ to any number in that range:

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

This has the expected typing:

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

which is derived using the subsumption rule (SUB).

Other simple examples of subtyping were given in the last chapter.

4.3.2 Subtyping type operators and families

Systems of *higher-order subtyping* extend the subtyping relation to type-constructors. They were invented to increase the scope of the subtyping model of object-oriented languages mentioned in Section 4.1. To date, most systems studied are variants of F^{ω} with subtyping, known as $F^{\omega}_{\leq}$ [Cardelli, 1990, Pierce and Turner, 1994, Steffen and Pierce, 1994, Compagnoni, 1995, Pierce and Pollack, 1992]. In these systems, one can declare a type variable ranging over type operators:

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

A related system which has dependent types instead of polymorphism is the calculus of subtyping and dependent types $\lambda P_{\leq}$ introduced by Aspinall and Compagnoni [1996]. In $\lambda P_{\leq}$ one can declare a variable ranging over type families:

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

In the first case, F ranges over constructors that map any subtype β of *nat* to a subtype of $List(\beta \times \beta)$; 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.

The higher-order subtyping systems have a pointwise rule for subtyping type operators constructed with λ:

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

Intuitively, $\lambda\alpha{:}K.A$ is a subtype of $\lambda\alpha{:}K.B$ if for every constructor C of kind K (or every element C of type K in $\lambda P_{\leq}$), the type $A[C/\alpha]$ is a subtype of the type $B[C/\alpha]$.

There is a corresponding rule for applications:

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

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 is an example using (SUB-APP):

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

(where $n : \mathit{nat}$ in the context). This derivation uses the conversion and transitivity rules.

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 Bruce and Mitchell [1992] and called the *HOPER model* in Compagnoni and Pierce [1996] and Steffen and Pierce [1994].

In λ_{Power}, there is no rule corresponding to (SUB-λ). Indeed it is impossible to prove anything with the form $\Gamma \rhd \lambda\alpha{:}K.\, A : \mathit{Power}(C)$. With power types, the rules above for higher-order subtyping would be harder to interpret semantically, at least because the interpretation of $\lambda\alpha{:}K.\, A : \mathit{Power}(C)$ would have to be considered pointwise rather than directly as a subset inclusion, so the meaning of *Power* in a term would depend on its context. This is a reflection that with higher-order subtyping, the subtyping relation is really a family of relations indexed by kinds.

Despite the lack of (SUB-λ) and (SUB-APP), it's pleasing to see that power types can express the same typings as higher-order subtyping. Here's an informal explanation of how. Suppose that α is constrained to be a subtype of a type-constructor H with domain K; this is exactly like asking α to be an element of $\Pi\alpha{:}K.\ \mathit{Power}(H\,\alpha)$, since each application of αM must be a subtype of $H\,M$. Using this "η-like" expansion for Π-types, higher-order bounded abstraction and quantification, and also bounded dependent type functions, can be reduced to abstraction and quantification over Π-types with power types in their codomain.

Instead of the variable declarations above, in λ_{Power} we could write:

$$F \quad : \quad \Pi\beta{:}\,\mathit{Power}(\mathit{nat}).\ \mathit{Power}(\mathit{List}(\beta \times \beta))$$

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

Now, to derive $G\,n \leq \mathit{Vec}_{\mathit{nat}}(5 * n)$ we need only a single use of (APP):

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

The substitution in the rule for dependent products takes the place of uses of conversion and transitivity in the other systems, so derivations in λ_{Power} are more direct.[4]

I believe that this is an original observation about power types. It provides further justification for using power types, in fact; 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.

4.3.3 λ_{Power} *as a logical framework*

λ_{Power} is related to the type system λP, which underlies the Edinburgh Logical Framework, LF [Harper et al., 1993].

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;
- Proof reuse: one proof term can prove several similar propositions.

[4]Although the *practical* effects on the differences in type-checking algorithms have not been fully investigated yet.

The desire for subtyping in LF was first mentioned by Mason [1987], to improve the representation for Hoare Logic. Later, Pfenning [1993] showed examples of proof-reuse through a restricted form of subtyping called *refinement types*. More recently, Aspinall and Compagnoni [1996] 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, Luo [1996] 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.

It's quite easy to see that λ_{Power} can be used in the same way as λP. Let υ be a single atomic type. Then declare a universe of types by writing:

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

Now we may 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 $\Gamma \rhd \Pi x{:}A.\, B : \mathit{Power}(\Pi x{:}A.\, \upsilon)$. But 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}.[5]

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 Pfenning [1993], Aspinall and Compagnoni [1996]. Where λ_{Power} goes beyond both these systems is in the possibility, for example, to refine the universe U.

ELF+

Gardner [1992] suggests refining the universe of LF. She defines an improved framework ELF+ which can distinguish between terms that represent pieces of object-level syntax, terms which represent proofs, and all other terms (such as those which represent rules of the logic). 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 [Breazu-Tannen et al., 1991]. 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.

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

ELF+ is defined as a Pure Type System but we can emulate it in λ_{Power} by declaring three subtypes of U:

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

In fact, we could equally well begin with three atomic types.

The idea is that types inhabiting Sort are the syntactic categories of the object logic, whilst types inhabiting Judge are its 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 Type is a spare universe to house so-called "extra constants" which are artefacts of the encoding with no correspondence in the object logic. In ELF+, Type also serves to house Π-abstractions of sorts and judgements.

Gardner gives examples of encodings which use these universes. Examples where power types are useful include higher-order logic [Gardner, 1992, Examples 4.2.3, 5.1.10] and the modal logic S4 [Gardner, 1992, Example 5.1.12]. The examples are carried out for LF in Avron et al. [1992]. I will highlight the improvements achieved with power types.

Higher-order logic. In the Church-style representation of higher-order logic (HOL), terms of the logic are simply typed, using types of the form:

$$\tau \;::=\; \iota \;\mid\; o \;\mid\; \tau \Rightarrow \tau$$

The HOL *types* are represented by elements in an LF type dom:

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

The HOL *terms* with domain τ are represented as elements of an LF type $\mathtt{obj}(\tau)$, where $\mathtt{obj} : \mathtt{dom} \to \mathtt{Type}$. In ELF+, dom and the mapping obj are shown to be artefacts of the encoding, because they inhabit Type, and obj is instead given the type:

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

to express that elements of $\mathtt{obj}(\tau)$ correspond with object logic syntax. In λ_{Power}, we can go a stage further, and remove obj altogether, by declaring:

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

Now we may imagine the mapping `obj` to be implicit. Because we no longer need to write `obj`, the representation of the logic becomes more concise, yet no less accurate. For example, the application term former is represented as:

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

instead of

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

In LF and ELF+, the proliferation of `obj` quickly pollutes large terms.

Although this example is simple, it is important to emphasise that it goes beyond the proposals of Pfenning [1993] and Aspinall and Compagnoni [1996] because dom is declared as a sub-*kind* of the universe `Sort`. Power types apply uniformly; other systems would have to be extended with subkinding to cope with this example.

Modal logic S4. The representation of Hilbert-style S4 in LF, due to Avron et al. [1992], 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:

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

The ELF+ typing of `valid` indicates that an expression `valid`(Φ) is an artefact 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 Longo et al. [1995].) A similar example by Pfenning [1993] 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 $\lambda P_{\leq}$. Both examples, once they make use of subtyping, no longer need the universe Type. 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.

4.4 Basic Properties

In this section I shall establish some simple 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 4.2 (Derivable rules).
These rules are derivable in λ_{Power}:

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

$$\frac{\Gamma \rhd A \leq B \qquad \Gamma \rhd B \leq C}{\Gamma \rhd A \leq C} \qquad \text{(SUB-TRANS)}$$

$$\frac{\Gamma \rhd A\ \mathit{type}}{\Gamma \rhd A = A} \qquad \text{(EQT-REFL)}$$

$$\frac{\Gamma \rhd B = A}{\Gamma \rhd A = B} \qquad \text{(EQT-SYM)}$$

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 shall be careful to distinguish derivable rules from those which are admissible but not derivable. This is because when considering the semantics we will treat some of the important admissible rules (e.g., substitution and thinning) as part of the system, making sure that they are valid in every model. These "important" admissible rules could be added to the presentation of the system as is done by other authors [Streicher, 1991, Pitts, 1996], but this spoils the direct proof of several meta-properties. So I stick with a presentation in which they are easily proved admissible.

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.

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

I shall use J as an additional meta-variable to range over the three judgements of the system, and $\Gamma \rhd J$ to indicate a judgement with context Γ. One derivation is "shorter" than another if it has fewer proof rules 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 componentwise, so that if $\Gamma \equiv x_1 : A_1, \ldots$ then $\Gamma[N/x] \equiv x_1 : A_1[N/x], x_2 : A_2[N/x], \ldots$. Substitution is also extended to judgements J componentwise.

The proofs of the next few propositions follow similar proofs given for $\lambda_{\leq\{\}}$ in Section 3.4 on page 55.

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

1. (Context 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 of contexts).
   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).
$\Gamma, x : A, \Gamma' \rhd J \quad \text{and} \quad \Gamma \rhd M : A \quad \Longrightarrow \quad \Gamma, \Gamma'[M/x] \rhd J[M/x].$

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

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

Proposition 4.7 (Formation).

1. $\Gamma \rhd \lambda x{:}A.\, M : C \quad \Longrightarrow \quad \Gamma \rhd A \ \mathit{type} \quad \text{and} \quad \exists B.\ \Gamma, x : A \rhd M : B.$

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

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

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

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. $\Gamma \rhd M : A \quad \Longrightarrow \quad \Gamma \rhd A \ \mathit{type}.$

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

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

The few basic equality rules of Figure 4.2 have some important admissible rules as consequences. These include congruence rules for the type constructors, because lambda-abstraction and application apply uniformly to types as well as terms. We also have 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).

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

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

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

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:

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

since from the first premise of (EQT-TRANS) we can then show $\Gamma \rhd A = B : \mathit{Power}(B)$ and by symmetry twice, $\Gamma \rhd B = C : \mathit{Power}(B)$, 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) demonstrates that the equality of types is an "absolute" notion, in the sense that the derivability of $\Gamma \rhd A = B : C$ is not affected by the type C when A and B are types. The term on the right-hand side of equality judgements in such cases merely ensures that equational deductions are combined properly, just as types in an equational theory for the simply typed λ-calculus do [Mitchell, 1990]. This contrasts with the case when A and B are non-type terms, when C can affect whether or not A and B are considered equal (see the example in Section 3.1).

This justifies using the derived judgement form $\Gamma \rhd A = B$. The semantics considered later reflects this aspect of the syntax, that type-equality is "absolute".

4.4.1 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 connection with reduction, known as *subject reduction*:

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

Subject reduction is an important practical property. It 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 subject reduction seems difficult to prove for λ_{Power}, and remains open. 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.

The key to proving subject reduction is to first establish a *generation principle* for the system, 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.

The formation proposition (Proposition 4.7) is a weak generation principle and says something about how derived judgements were built up. But it is not strong enough. To tackle the meta-theory of λ_{Power} we need a stronger principle which garners a link between the type C of a judgement $\Gamma \rhd M : C$ and the judgements concerning subterms of M asserted to exist. For example, for the λ-case, it might be something like 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 an application of (λ) to derive $\lambda x{:}A.\,M : \Pi x{:}A.\,B$, there can be a series of uses of subsumption (SUB) and conversion (CONV) through which $\Pi x{:}A.\,B$ mutates into C:

$$\dfrac{x : A \rhd M : B}{\lambda x{:}A.\,M : \Pi x{:}A.\,B}$$
$$\vdots$$
$$\dfrac{\lambda x{:}A.\,M : C_j \qquad C_j \leq C_{j+1}}{\lambda x{:}A.\,M : C_{j+1}}\ (\text{SUB})$$
$$\vdots$$
$$\dfrac{\lambda x{:}A.\,M : C_k \qquad\qquad C_k = C_{k+1}}{\lambda x{:}A.\,M : C_{k+1}}\ (\text{CONV})$$
$$\vdots$$
$$\lambda x{:}A.\,M : C$$

The two cut-like rules (SUB) and (CONV) make it tricky to prove the statement above by a direct induction, because arbitrary terms can appear in the intervening typings. To "join up" the C_i's we need to appeal to the generation principle being proved. Actually, it is worse than this, because the rule (REFL) may also appear in the derivation introducing further detours, so the putative statement of λ-generation given above needs altering.

The traditional syntactic way to handle this situation is to by reformulating the system in a more syntax-directed fashion, 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 [Curien and Ghelli, 1992, Compagnoni, 1995], including the first dependent type system in Aspinall and Compagnoni [1996]. Unfortunately it is hard to extend the techniques there directly to λ_{Power}. The sticking point is bounded operator abstraction which makes it difficult to prove lemmas about substitution in the syntax-directed system before having proved other properties which in turn depend on substitution. For $\lambda P_{\leq}$ in Aspinall and Compagnoni [1996], we managed to find a judicious order in which to prove these properties.

A new proof technique has been developed recently by Compagnoni and Goguen [1997] which avoids this problem. 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 4.6 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).
$\Gamma, x : A, \Gamma' \rhd J \quad \text{and} \quad x \notin FV(\Gamma', J) \quad \Longrightarrow \quad \Gamma, \Gamma' \rhd J.$

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

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

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.

4.4.2 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). A comparison between λ_{Power} and the related work appears in Section 4.9.1.

Lack of type-formation judgement

Typically, dependent type theories have a judgement form

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

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 $\Gamma \rhd A\ \mathit{type}$ as a tacit premise.

In λ_{Power}, typing judgements of the form $\Gamma \rhd A : \mathit{Power}(B)$ 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; $A : \mathit{Power}(A)$ 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 is much easier to present the syntax in this way 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 have a rule:

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

which collapses the two notions anyway.

Although 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.

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. Luo [1994] calls this confusion "multiple-identity". It is a hang-over 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 Luo, 1994] 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 a rule:

$$\frac{M : \mathit{Power}(A)}{T(M) \leq A}$$

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 Barendregt [1992].) 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 [Luo, 1994].

How deeply nested do power types get? In Section 4.3.3 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 Figures 4.1 and 4.2 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 [Geuvers and Werner, 1994]. The meta-theory 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 [Geuvers, 1993]). For semantic presentations, the typical method of attack has been via a model construction [Streicher, 1991, Goguen, 1994].

Pure Type Systems [Barendregt, 1992] are defined with a syntactic presentation using the second rule above. The original presentation of ASL+ [Sannella et al., 1992] is similar, but uses split rules 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 4.4.1.

4.5 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 which is a necessary condition for typability in the full system. It is easy to check rough typability because we never need to test equality of terms, so we have *static* type-checking for rough types. More over, two pre-terms which are in the full typing relation of λ_{Power} must have related rough types, and two terms which are equal in the equational theory must have the same rough type.

The idea of rough type-checking in λ_{Power} comes from Sannella et al. [1992][6], where it was also suggested that rough types could be used to give a

[6]Rough types were called "types" in Sannella et al. [1992], but using this name we couldn't call anything in the full system a "type". Rough types were called "kinds" in the early presentation of my work in [Aspinall, 1995b], but this name confused some people. Another possibility is "simple type" which suggests the removal of dependency and relationship to λ^{-}. I shall stick to "rough types" here because it gives extra intuition and relates to the explanation in Sannella et al. [1992] where a term is called "roughly well

foundationally-secure semantics to the language. The model definition given in Section 4.6 uses rough types to do this.

A further application of rough types is the proof of strong normalization for λ_{Power}, which follows from strongly normalization for roughly-typable terms. This proof is given in Section 4.8.

4.5.1 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 inductively by the grammar:

$$\boldsymbol{Ty} \ ::= \ \mathcal{K} \ \mid \ \boldsymbol{Ty} \Rightarrow \boldsymbol{Ty} \ \mid \ P(\boldsymbol{Ty})$$

We write just $\boldsymbol{Ty}$ for short when the set $\mathcal{K}$ is understood.

These are the types of the simply-typed λ-calculus extended with a power type constructor. We use $\tau, \upsilon, \ldots$ to range over $\boldsymbol{Ty}$.

There are two rough typing judgements:

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

The judgements are defined inductively by the rules in Figure 4.3 on the next page. 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 way to understand rough typing rules is 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 $P(\tau \Rightarrow \upsilon)$, which indicates that it denotes a collection of functions.

Example 4.14. To illustrate rough typing, recall the example context Γ_{PERM} from Section 4.3.1 on page 91. We can derive these rough typings:

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

formed" if it has a rough type.

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

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

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

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

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

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

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

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

Figure 4.3: Rough typing for λ_{Power}

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

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:

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

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

In Theorem 4.19 on the next page, we prove that typability in the full calculus guarantees rough typability. From the above example we can see that the converse fails, since we have

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

but this term cannot be typed in the rules of Figure 4.1. (To prove this we would need to use a generation principle or model construction).

Although rough typing judgements are not generally 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 4.3 and Figure 4.1 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 Sannella and Tarlecki [1991].

4.5.2 Properties of rough typing

It is easy to establish meta-properties of the rough-typing system because the types are non-dependent and subtyping has been removed.

Proposition 4.15 (Properties of rough typing).

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

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 4.3, 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.[7] □

Proposition 4.16 (Thinning and substitution for rough typing).
1. *If* $\Gamma \blacktriangleright M \Rightarrow \tau$ *and* $\Gamma \subseteq \Gamma'$ *with* $\blacktriangleright \Gamma'$, *then* $\Gamma' \blacktriangleright M \Rightarrow \tau$.

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

Proof Each part by induction on derivations. □

Proposition 4.17 (Strengthening for rough typing).
If $\Gamma, x : A, \Gamma' \blacktriangleright M \Rightarrow \tau$ *and* $x \notin FV(\Gamma'), FV(M)$, *then* $\Gamma, \Gamma' \blacktriangleright M \Rightarrow \tau$.

Proof By simultaneous induction also for the context rough-typing judgement, using free-variable properties of roughly-typable contexts (analogous to part 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. This allows us to show the subject reduction property without any fuss.

Proposition 4.18 (Subject reduction for rough typing).
If $\Gamma \blacktriangleright M \Rightarrow \tau$ *and* $M \twoheadrightarrow_{\beta\eta} M'$, *then* $\Gamma \blacktriangleright M' \Rightarrow \tau$ *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}, which was claimed at the beginning of this section. Part 4 is just needed to make the induction go through.

Theorem 4.19 (Agreement of rough typing).
1. *If* $\rhd \Gamma$ *then* $\blacktriangleright \Gamma$.

[7]This assumes that we can determine the syntactic identity of two atomic types, i.e., whether $\kappa \equiv \kappa'$.

2. *If $\Gamma \rhd M : A$ then for some $\tau \in \mathbf{Ty}$, $\Gamma \blacktriangleright M \Longrightarrow \tau$ and $\Gamma \blacktriangleright A \Longrightarrow \mathsf{P}(\tau)$.*

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

4. *If $\Gamma \rhd A : \mathsf{Power}(B)$ then for some $\tau \in \mathbf{Ty}$, $\Gamma \blacktriangleright A \Longrightarrow \mathsf{P}(\tau)$*

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

So if $\Gamma \rhd M : A$ in λ_{Power}, M and A must necessarily be typable with rough types, and moreover, the rough type of A is $\mathsf{P}(\tau)$ where τ is the rough type of M.

4.5.3 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. This can be difficult with subtyping and bounded type variables; see Section 4.9.1 on page 135 for more discussion.

Different atomic types. A generalization of the connection between λ_{Power} and its rough types would be to allow different sets of atomic types, and consider a mapping from $\mathcal{K}$ in λ_{Power} to types over a set $\mathcal{RK}$ 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 $\mathsf{P}(\tau)$ 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 [1993]. 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 $\mathsf{P}(\tau)$ as an abbreviation for

$\tau \times \tau \Rightarrow \mathsf{Prop}$. This interpretation is closely connected with models considered in Section 4.6, and also with the understanding of subtyping as intuitionistic implication explained by Longo et al. [1995].

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 [Geuvers and Nederhof, 1991, Harper et al., 1993]. Strong normalization for λ_{Power} is proved in Section 4.8 using rough typing.

4.6 Semantics

In this section I shall give an environment model definition for λ_{Power} and some examples of models. This begins in Section 4.6.1 on page 117. Before that I shall discuss some of the motivations for doing this, and how the model definition was arrived at.

Motivations

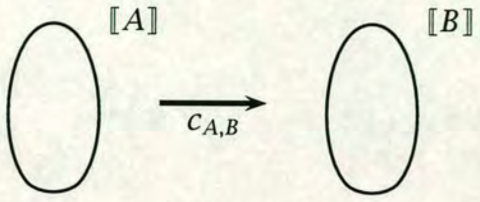
For constructive type theories, a semantical interpretation is often a secondary construction, perhaps a tool to prove consistency or strong normalization. The important thing for such theories is the syntax. But applying λ_{Power} to ASL+, we have an intended model where atomic types denote classes 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 [1995b], changing the model given in Sannella et al. [1992] and using rough typing. That model is an instance of the model definition given for $\lambda_{\mathrm{ASL+}}$ in the next chapter, which extends the model definition for λ_{Power} given here.

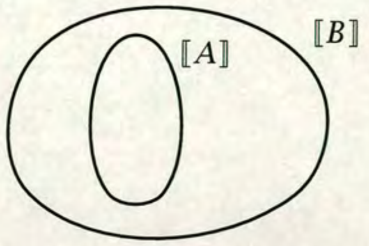
Containment versus coercion

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 [Cardelli and Longo, 1991], or we reconstruct coercions by a translation process [Breazu-Tannen et al., 1991]. 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 Curien and Ghelli [1992] for this, or [Mitchell, 1996, 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.

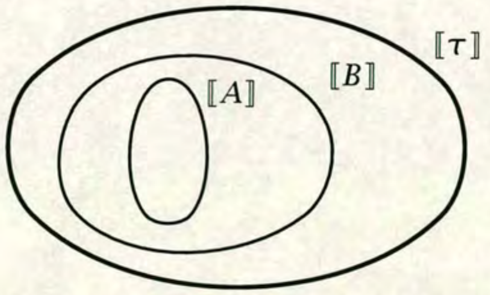
With a containment semantics there is no problem of coherence, but there is a fundamental difficulty with the contravariant rule for subtyping Π-types.

In the syntax we have that $int \to int \leq nat \to int$, but this does not hold as a set-theoretic inclusion; $\mathbf{Z} \to \mathbf{Z} \nsubseteq \mathbf{N} \to \mathbf{Z}$ when the semantic $\to$ denotes set-theoretic function space. This is usually solved by interpreting $nat \to int$ 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. 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, as was done for the model of $\lambda_{\leq\{\}}$ in Section 3.6. But it is surprising to base a semantics for that calculus and even for $\lambda_{\leq}$ on a model of the untyped λ-calculus; λ-models contain many more functions than can be expressed in the typed language being modelled. Yet to date this is the approach that has been followed most often in the literature.[8] For λ_{Power} I shall stay closer to the much 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 $P(\tau)$, 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 $P(\tau_C \Rightarrow \tau_D)$, 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,

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

[8]It is perhaps more excusable once the language is enriched with polymorphism and other constructs.

The main reason that I choose 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 algebraic signatures correspond to the atomic types used to build the rough types for $\lambda_{\mathsf{ASL+}}$ in Section 5.2.

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. Section 3.1 described the stratified equality between terms in a subtyping system. 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.

Plan

The model definition for λ_{Power} is given in stages, in a standard way. 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 4.6.1. 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 4.6.2. Finally we give the model definition itself, in Section 4.6.3, which is a set of axioms that must be satisfied by an interpretation.

4.6.1 Structures

A λ_{Power} applicative structure is similar to an ordinary typed-applicative structure for $\lambda^{\rightarrow}$ [Mitchell, 1990, see e.g.,]. It provides semantic domains for every rough type of λ_{Power}. The domains are sets.

Definition 4.20 (λ_{Power} applicative structure).
A λ_{Power} *applicative structure* $\mathcal{D} = \langle \mathcal{D}, Const, App \rangle$ consists of:

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

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

Of course there are many λ_{Power} structures; interesting examples are the full type hierarchy, a variant of it with partial functions, and the term structure. The second example structure will be applied later in a model for ASL+ (in Chapter 5). The term structure will be a good test for the model definition given later, and has potential applications not studied further here. The simple case of the full type hierarchy with total functions is the most obvious instance of the definitions.

Notation 4.21. Given a set S, I write $\text{REL}(S)$ for the set of relations on S, $\text{REL}(S) =_{\text{def}} Pow(S \times S)$. If $R \in \text{REL}(S)$, then $dom(R) = \{ a \mid a \mathrel{R} b \}$. A relation is a *partial equivalence* (PER) if it is symmetric and transitive. I write $\text{PER}(S)$ for the set of PERs on S. The notation $a \mapsto f(a)$ is used to stand for the function mapping a to $f(a)$, in other words, λ-abstraction in the meta-language.

Example 4.22 (Full hierarchy structure). Given a family of sets and PERs $C = \{ C_{\kappa}, R_{\kappa} \in \text{PER}(C_{\kappa}) \}_{\kappa \in \mathcal{K}}$, the *full hierarchy* $\mathcal{F}_C$ on C is defined by

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

□

In the full hierarchy structure, we define the set $\mathcal{F}^{P(\tau)}$ to be the set of all relations over $\mathcal{F}^{\tau}$, rather than the set of all PERs. The reason for this is technical: because the interpretation in the full structure (Example 4.28) is defined over rough types, the type-constructors are not guaranteed to construct PERs. PERs *are* guaranteed by the soundness result for the full calculus, however. Using $\mathrm{REL}(\mathcal{F}^{\tau})$ is a design choice: instead we could adjust the interpretation (in Example 4.28) 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 4.23 (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

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

where $\diamond$ is a distinct token, not in any C_{κ}. □

Example 4.24 (Term structure). The *term structure* $\mathcal{T}_{\Gamma}$ of roughly-typed terms in a pre-context Γ is defined by

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

□

4.6.2 Environments and interpretations

To define the interpretation of terms, we need to interpret free variables declared in the context. For each roughly-typable context Γ, a semantic domain

$\mathcal{D}^{\Gamma}$ is defined by induction on Γ:

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

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

Given a Γ-environment $\eta \in \mathcal{D}^{\Gamma}$, we define projections for the variables of Γ by:

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

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

We can define thinning between environments using the projection notation. If $\Gamma_1 \subseteq \Gamma_2$ and we have $\eta_1 \in \mathcal{D}^{\Gamma_1}$ and $\eta_2 \in \mathcal{D}^{\Gamma_2}$ then we write ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$ iff

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

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

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

Unlike a partial function environment mapping variables to $\bigcup \{ \mathcal{D}^{\tau} \}_{\tau \in \boldsymbol{Ty}}$, this tupled form of environment comes with 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 categorical semantics of simply-typed λ-calculus in (set-like) CCCs. Tuples are a rather concrete construction: more abstractly, contexts could be treated in the same way as types.

Definition 4.25 (λ_{Power} interpretation).
An λ_{Power} *interpretation* in $\mathcal{D}$ consists of:

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

The meaning function in an interpretation is indexed by a rough type and a context; we use the judgement notation in $[\![\Gamma \blacktriangleright M \Rightarrow \tau]\!]_\eta$ 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 $[\![M]\!]_\eta$.

The mapping Rel^τ in an interpretation characterises the behaviour (or *extension*) of the elements denoting types, just as App characterises the extension of the elements denoting functions. It is part of the interpretation so that we could consider different "views" of types within the same applicative structure. If $a \in \mathcal{D}^{P(\tau)}$ I shall write R_a as shorthand for the relation given by $\mathit{Rel}^\tau(a)$.

The definition of interpretation does not require a priori that $\mathit{Rel}^\tau(a)$ is a partial equivalence relation, for the reason outlined before. Instead it will be a consequence of the soundness theorem that any well-formed type of λ_{Power} in fact denotes a PER. This differs from the concrete model instance for $\lambda_{\leq\{\}}$ given in the last chapter, where the interpretation was a partial definition which proved to be well-defined on well-typed terms; here we assume that any interpretation function is well-defined on all *roughly-typed* terms.

We show how the structures of Examples 4.22–4.24 can be extended to interpretations in the next section.

4.6.3 Models

Before the definition, here are two constructions on relations. Let $\mathcal{D}$ be a λ_{Power} interpretation. Given $R \in \mathrm{REL}(\mathcal{D}^\tau)$ and $G \in \mathit{dom}(R) \to \mathrm{REL}(\mathcal{D}^\upsilon)$, define

- $\Pi(R,G) \in \mathrm{REL}(\mathcal{D}^{\tau\Rightarrow\upsilon})$ by:

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

- $\mathcal{P}wr(R) \in \mathrm{PER}(\mathcal{D}^{P(\tau)})$ by:

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

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

Fact 4.26. 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 4.27 (λ_{Power} environment model).
A λ_{Power} *environment model* for a structure $\mathcal{D}$ is an interpretation for $\mathcal{D}$ such that the following conditions are satisfied.
For all roughly-typable contexts Γ and all $\eta \in \mathcal{D}^{\Gamma}$,

CONST $\quad [\![\kappa]\!]_{\eta} = \mathit{Const}(\kappa).$

CONST2 $\quad R_{[\![\kappa]\!]_{\eta}} \in \mathrm{PER}(\mathcal{D}^{\kappa}).$

VAR $\quad [\![x]\!]_{\eta} = \eta^{\Gamma}(x).$

APP $\quad [\![M\,N]\!]_{\eta} = \mathit{App}([\![M]\!]_{\eta}, [\![N]\!]_{\eta}).$

FAMILY $\quad$ 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 $\quad R_{[\![\mathit{Power}(C)]\!]_{\eta}} = \mathcal{P}wr(R_{[\![C]\!]_{\eta}}).$

In the above axioms we assume that for some τ, υ, $\Gamma \blacktriangleright x \implies \tau$, $\Gamma \blacktriangleright \Pi x{:}A.B \implies \mathsf{P}(\tau \Rightarrow \upsilon)$ and $\Gamma \blacktriangleright \mathit{Power}(C) \implies \mathsf{P}(\mathsf{P}(\tau))$.

ABS If $\Gamma \blacktriangleright A \implies \mathsf{P}(\tau)$, $\Gamma, x : A \blacktriangleright B \implies \mathsf{P}(\upsilon)$, and $\Gamma \blacktriangleright M, N \implies \tau$, then

$$\forall d, e.\ \ d\ R_{[\![A]\!]_{\eta}}\ e \implies [\![M]\!]_{\langle \eta, d\rangle}\ \ R_{[\![B]\!]_{\langle \eta, d\rangle}}\ \ [\![N]\!]_{\langle \eta, e\rangle}$$

implies

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

THIN Let Γ_1, Γ_2 be roughly-typable contexts with Φ a renaming on $\mathit{Dom}(\Gamma_1)$, $\Phi(\Gamma_1) \subseteq \Gamma_2$, $\eta_1 \in \mathcal{D}^{\Gamma_1}$ and $\eta_2 \in \mathcal{D}^{\Gamma_2}$. We require that

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

whenever $\Gamma \blacktriangleright M \implies \tau$ and ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$.

SUBST Let $\Gamma_1 \equiv \Gamma, x : A, \Gamma'$, and $\Gamma_2 \equiv \Gamma, \Gamma'[N/x]$ where $\Gamma \blacktriangleright N \implies \upsilon$ and $\Gamma \blacktriangleright A \implies \mathsf{P}(\upsilon)$. Let $\eta_1 \in \mathcal{D}^{\Gamma_1}$ and $\eta_2 \in \mathcal{D}^{\Gamma_2}$. We require that

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

whenever $\Gamma_1 \blacktriangleright M \implies \tau$, ${\eta_1}^{\Gamma_1}(x) = [\![\Gamma \blacktriangleright N]\!]_{\eta_1|_x} \in \mathit{Rel}([\![\Gamma \blacktriangleright A]\!]_{\eta_1|_x})$ and ${\eta_1}^{\Gamma_1} \subseteq {\eta_2}^{\Gamma_2}$. □

The axioms CONST, VAR, APP define the meaning function in these cases, and are standard in the model-definition for untyped and simply-typed λ-calculus [Meyer, 1982, Mitchell, 1990]. The axiom CONST2 requires that atomic types are interpreted as 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 two fundamental properties that any model should satisfy. The substitution property need only be satisfied when the environment is updated with a value which lies in the domain of the PER which interprets the type A in the context. Proposition 4.16 ensures that the conditions THIN and SUBST are meaningful.

The soundness proof for models is given in Section 4.7. In the rest of this section each example structure is extended to a model.

Full hierarchy model

Example 4.28. An interpretation in the full structure can be defined by:

$$\mathit{Rel}(A) \;=\; A$$
$$[\![\Gamma \blacktriangleright x \Rightarrow \tau\ ]\!]_\eta = \eta^n(x)$$
$$[\![\Gamma \blacktriangleright \kappa \Rightarrow P(\kappa)]\!]_\eta \;=\; R_\kappa$$
$$[\![\Gamma \blacktriangleright \lambda x{:}A.\,M \Rightarrow \tau \Rightarrow \upsilon]\!]_\eta \;=$$

the function $f : \mathcal{D}^\tau \to \mathcal{D}^\upsilon$ defined by

$$f(n) = [\![\Gamma, x : A \blacktriangleright M \Rightarrow \upsilon]\!]_{\langle \eta, n\rangle}$$

for all $n \in \mathcal{D}^\tau$.

$$[\![\Gamma \blacktriangleright M N \Rightarrow \upsilon]\!]_\eta \;=\; \mathit{App}([\![\Gamma \blacktriangleright M \Rightarrow \tau \Rightarrow \upsilon]\!]_\eta, [\![\Gamma \blacktriangleright N \Rightarrow \tau]\!]_\eta)$$
$$[\![\Gamma \blacktriangleright \Pi x{:}A.\,B \Rightarrow P(\tau \Rightarrow \upsilon)]\!]_\eta \;=\; \Pi(R_{[\![A]\!]_\eta}, a \mapsto R_{[\![B]\!]_{\langle \eta, a\rangle}})$$
$$[\![\Gamma \blacktriangleright \mathit{Power}(A) \Rightarrow P(P(\tau))]\!]_\eta \;=\; \mathcal{P}wr(R_{[\![A]\!]_\eta})$$

In this structure, it is not necessarily the case that $R_{[\![\Pi x:A.\,B]\!]_\eta}$ is a PER, since the uniformity condition that $a \;\; R_{[\![A]\!]_\eta} \;\; b \;\; \Rightarrow \;\; R_{[\![B]\!]_{\langle \eta, a\rangle}} = R_{[\![B]\!]_{\langle \eta, b\rangle}}$ may fail. For example, if $B \equiv zx$, the "rough-soundness" requirement that $\eta(z) \in \mathcal{D}^{P(\tau \Rightarrow P(\upsilon))}$ imposes no restriction relating the value of z at one element of $\mathcal{D}^\tau$ to another. This explains the technical need to generalize to relations.

Lemma 4.29. *The interpretation defined in Example 4.28 is a model of* λ_{Power}.

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

In the interpretation of Example 4.28, 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 4.30. An interpretation in the partial structure can be defined by:

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

$$[\![\Gamma \blacktriangleright x \implies \tau]\!]_{\eta} \quad = \quad \eta^{\eta}(x)$$

$$[\![\Gamma \blacktriangleright \kappa \implies P(\tau)]\!]_{\eta} \quad = \quad R_{\kappa}$$

$$[\![\Gamma \blacktriangleright \lambda x{:}A.\,M \implies \tau \Rightarrow \upsilon]\!]_{\eta} \quad =$$

the partial function $f : \mathcal{D}^{\tau} \rightharpoonup \mathcal{D}^{\upsilon}$ defined by

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

for all $n \in \mathcal{D}^{\tau}$.

$$[\![\Gamma \blacktriangleright M\,N \implies \upsilon]\!]_{\eta} \quad = \quad App([\![\Gamma \blacktriangleright M \implies \tau \Rightarrow \upsilon]\!]_{\eta}, [\![\Gamma \blacktriangleright N \implies \tau]\!]_{\eta})$$

$$[\![\Gamma \blacktriangleright \Pi x{:}A.\,B \implies P(\tau \Rightarrow \upsilon)]\!]_{\eta} \quad = \quad \Pi(R_{[\![A]\!]_{\eta}}, a \mapsto R_{[\![B]\!]_{\langle\eta,a\rangle}})$$

$$[\![\Gamma \blacktriangleright Power(A) \implies P(P(\tau))]\!]_{\eta} \quad = \quad \mathcal{P}wr(R_{[\![A]\!]_{\eta}})$$

Lemma 4.31. *The interpretation defined in Example 4.30 is a model of* λ_{Power}.

Proof Similar to the proof of Lemma 4.29. □

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 $Rel([\![A]\!]_{\eta})$. 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 Streicher [1991][9]. 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

[9]and in fact independently by Sannella et al. [1992] for ASL+.

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 [1996] 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.)

Unfortunately, the partial function model of Example 4.30 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 4.38 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:

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

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 4.38 could be replayed to establish the correctness property above.

Term model

The final example is the term model, which 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.

Example 4.32. An interpretation in the term structure $\mathcal{T}_\Gamma$ can be defined by:

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

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

It seems difficult to prove directly that the model conditions are satisfied by the term model. In particular, showing the right-to-left direction of the equalities in FAMILY and SUBSET are satisfied calls for some of the advanced meta-properties of the syntax mentioned in Section 4.4.1. (This is a shame because I had hoped that a term model construction would give some help in establishing those very properties!)

Conjecture 4.33. *The interpretation defined in Example 4.32 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. □

4.7 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, $\mathit{Rel}([\![A]\!]_\eta)$ is a partial equivalence relation on $\mathcal{D}^{\tau}$, where $\Gamma \blacktriangleright A \Rightarrow P(\tau)$.

However, we can only expect soundness if the environment η satisfies the context in a suitable sense. The interpretation of a context Γ is defined by combining the relations which interpret its component types.

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:

$$p \ \Sigma(R,G)\ q \qquad \text{iff} \qquad \begin{cases} \pi_1(p)\ R\ \pi_1(q) \\ \pi_2(p)\ G(\pi_1(p))\ \pi_2(q) \end{cases}$$

Fact 4.34. 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 4.35 (Interpretation of contexts).

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

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

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

Fact 4.36. If Γ is roughly-typable and $\eta_1 \;[\![\Gamma]\!]\; \eta_2$, then it follows that

$$\eta_1^\Gamma(x) \quad R_{[\![\Gamma|_x \blacktriangleright \Gamma(x) \Rightarrow P(\tau)]\!]_{\eta_1|_x}} \quad \eta_2^\Gamma(x)$$

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

Lemma 4.37. *Suppose that* $R_{[\![A]\!]_\eta} \in \mathrm{PER}(\mathcal{D}^\tau)$ *and that* $d \;\; R_{[\![A]\!]_\eta} \;\; e \;\; \Rightarrow R_{[\![B]\!]_{\langle\eta,d\rangle}} = R_{[\![B]\!]_{\langle\eta,e\rangle}} \in \mathrm{PER}(\mathcal{D}^\upsilon)$ *for all* d, e. *Then the following properties hold in any model:*

WEAK-EXT $\quad \forall d,e.\;\; d \; R_{[\![A]\!]_\eta} \; e \Rightarrow [\![M]\!]_{\langle\eta,d\rangle} \;\; R_{[\![B]\!]_{\langle\eta,d\rangle}} \;\; [\![N]\!]_{\langle\eta,e\rangle}$

implies

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

ETA $\quad \forall d,e.\;\; d \; R_{[\![A]\!]_\eta} \; e \Rightarrow \mathsf{App}([\![M]\!]_\eta, d) \;\; R_{[\![B]\!]_{\langle\eta,d\rangle}} \;\; \mathsf{App}([\![N]\!]_\eta, e)$

implies

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

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

Theorem 4.38 (Soundness for models).

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

Proof Simultaneously by induction on derivation heights, using the model conditions.

At a couple of points in the proof, we assume that any derivation of $\Gamma \,\rhd\, M : A$ or $\Gamma \,\rhd\, M = N : A$ has a shorter derivation of $\Gamma \,\rhd\, A$ *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 $\Gamma \blacktriangleright A \implies P(\tau)$. The assumption is justified because any derivation has a corresponding derivation in a modified system with "strong" rules with subderivations $\Gamma \rhd A$ *type* in the premises. It is easy to argue that the system with strong rules is equivalent, using Proposition 4.7 (Troelstra [1987] gives details). 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.

Now we consider each of the cases in detail. The part 2 cases for (λ) and (APP) are special cases of part 3 for (EQ-λ) and (EQ-APP), respectively.

This proof ends on page 132.

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

   Case (EXTEND): We have $\rhd \Gamma', x : A$. By the induction hypothesis and the premise $\Gamma' \rhd A$ *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 $\Gamma' \blacktriangleright A \implies P(\tau)$. By the induction hypothesis for the shorter derivation of $\rhd \Gamma'$, $[\![\Gamma']\!] \in \mathrm{PER}(\mathcal{D}^{\Gamma})$. So by Fact 4.34, $[\![\Gamma', x : A]\!] \in \mathrm{PER}(\mathcal{D}^{\Gamma})$ as required.

2. **Case** (ATOMIC): We have $\kappa : \mathit{Power}(\kappa)$, so by CONST, we must show that

   $$\mathit{Const}(\kappa) \ R_{[\![\mathit{Power}(\kappa)]\!]_{\eta_1}} \ \mathit{Const}(\kappa)$$

   which by SUBSET holds iff

   $$R_{\mathit{Const}(\kappa)} = R_{\mathit{Const}(\kappa)} \in \mathrm{PER}(\mathcal{D}^{\kappa}) \quad \text{and} \quad R_{\mathit{Const}(\kappa)} \subseteq R_{\mathit{Const}(\kappa)}$$

   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 Fact 4.36 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

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

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

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

which by SUBSET holds iff

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

and

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

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

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

hence by FAMILY,

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

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

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

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

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

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

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

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

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

for all $a \in R_{[\![A]\!]_{\eta_1}}$, so by (4), (5), FAMILY and Fact 4.26 the equation (1) holds.

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

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

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

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

so

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

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

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

hence

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

as required.

Case (*Power*): We must show

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

which by SUBSET holds iff

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

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

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 $s \; R_{[\![Power(A)]\!]_{\eta_1}} \; t$. 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 $\Gamma \rhd A$ *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

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

   from which the result follows by FAMILY.

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

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

   Using the induction hypothesis for the premises, we have

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

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

and so by the axiom FAMILY, we have

$$App([\![M]\!]_{\eta_1}, [\![N]\!]_{\eta_1}) \; R_{[\![B]\!]_{\langle \eta_1, [\![N]\!]_{\eta_1} \rangle}} \; App([\![M']\!]_{\eta_2}, [\![N']\!]_{\eta_2})$$

But using SUBST,

$$[\![B]\!]_{\langle \eta_1, [\![N]\!]_{\eta_1} \rangle} \quad = \quad [\![B[N/x]]\!]_{\eta_1}$$

so we are done.

Case (EQ-β): We must show

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

By the induction hypothesis,

$$[\![\Gamma, x : A \rhd M]\!]_{\langle \eta_1, d \rangle} \; R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_1, d \rangle}} \; [\![\Gamma, x : A \rhd M']\!]_{\langle \eta_2, e \rangle}$$

for all d, e such that $d \; R_{[\![\Gamma \rhd A]\!]_{\eta_1}} \; e$, and

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

using SUBST,

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

and

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

By the ABS axiom,

$$App([\![\lambda x{:}A.\, M]\!]_{\eta_1}, [\![N]\!]_{\eta_1}) \; R_{[\![\Gamma, x:A \rhd B]\!]_{\langle \eta_1, [\![N]\!]_{\eta_1} \rangle}} \; [\![M']\!]_{\langle \eta_2, [\![N']\!]_{\eta_2} \rangle}$$

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

Case (EQ-η): We must show

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

By FAMILY, it suffices to show that

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

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

As an extension of this theorem, all of the admissible rules shown in Section 4.4 are sound in any model; details are omitted.

Here is the special case of Theorem 4.38 which expresses soundness of the typing judgement.

Corollary 4.39 (Soundness of Typing).
If $\Gamma \rhd M : A$, then for all η, $\eta \in R_{[\Gamma]} \quad \Rightarrow \quad [\![M]\!]_\eta \in R_{[A]_\eta}$.

4.8 Strong Normalization

In this section I show that $\beta\eta$-reduction is strongly normalizing on roughly-typed terms. By Theorem 4.19 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 [Barendregt, 1992, Luo, 1994, see, for example], with amendments for power types. We use a slightly extended language of terms,

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

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.[10] The new constructor has a rough typing rule:

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

Now some basic definitions.

Definition 4.40.

- A term is a *canonical term* if it has one of the forms: κ, $\lambda x{:}A.\,M$, $\Pi x{:}A.\,B$, or $\mathit{Power}(A)$.
- A term is a *base term* if it 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.

[10]For the cognoscenti, the reduction relation includes the replacement $\lfloor \mathit{Power}(A) \rfloor \longrightarrow_p A$. 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 [Curien and Ghelli, 1992, Compagnoni, 1995, Steffen and Pierce, 1994] are re-expressed for power types.

- SN is 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 4.41 (Saturated sets).
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 4.6.1 for notation, although the language is extended here.

Definition 4.42.

- The term applicative structure S is defined by:

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

- Let $\Gamma \blacktriangleright M \implies \tau$ and $\eta \in S^{\Gamma}$. Define $[\![\Gamma \blacktriangleright M \implies \tau]\!]_{\eta} = \eta^{\Gamma}(M)$.

I haven't bothered to define App and Const because the λ_{Power} model axioms aren't needed. The second lemma below says that $[\![\Gamma \blacktriangleright M \implies \tau]\!]_{\eta}$ satisfies the "rough soundness" requirement for meaning functions given as the first part of Definition 4.25. This yields the strong normalization result.

Lemma 4.43. *For all τ, the set S^{τ} is saturated.*

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

1. Suppose $M \in S^{P(\upsilon)}$. Then $\lfloor M \rfloor \in S^{\upsilon}$ and is SN by the induction hypothesis. Hence M is SN.
2. Let M be a SN base term. Then so too is $\lfloor M \rfloor$, which establishes $M \in S^{P(\upsilon)}$.

3. Suppose that $M \longrightarrow_{wh} M'$ and $M' \in S^{P(\upsilon)}$. 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 $M \in S^{P(\upsilon)}$. □

Lemma 4.44. *Let $\Gamma \blacktriangleright M \implies \tau$. For all $\eta \in S^{\Gamma}$, $[\![\Gamma \blacktriangleright M \implies \tau]\!]_{\eta} \in S^{\tau}$.*

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

Theorem 4.45 (Strong normalization for roughly-typed terms).
If $\Gamma \blacktriangleright M \implies \tau$ then M is $\beta\eta$-strongly normalizing.

Proof Immediate by Lemmas 4.43 and 4.44. □

Corollary 4.46 (Strong normalization for λ_{Power}).
If $\Gamma \vartriangleright M : A$, then M is $\beta\eta$-strongly normalizing.

Proof Immediate by Theorem 4.19. □

4.9 Discussion

This section contains a comparison between λ_{Power} and related type systems, and a note about a simplified formulation of λ_{Power}. Section 4.10 concludes this chapter with a summary.

4.9.1 Comparison with other systems

Cardelli. The closest relative to λ_{Power} is Cardelli's power type system introduced in Cardelli [1988]. That system was not studied in great detail, however, and its sheer complexity (number of typing features) means that it has some necessarily tricky aspects — type-checking is undecidable and every type is inhabited.

λ_{Power} is a fragment of Cardelli's system, with some differences:

- the rules for deriving equality are made explicit as an equality judgement; they were described only informally in Cardelli [1988].
- there is no type formation judgement (see remarks in Section 4.4.2).

LF and higher-order subtyping systems. Section 4.3 has already compared λ_{Power} against some related systems, including higher-order subtyping systems based on $F^{\omega}_{\leq}$, see Section 4.3.2 on page 92. At first appearance, λ_{Power} is a simplification of these systems, since subtyping of type-constructors is achieved indirectly, so we need fewer rules. But because of its uniformity, there is another sense in which λ_{Power} is more complicated.

In the formulations of $F^{\omega}_{\leq}$ successfully studied to date, type-constructors are unbounded. For example, it is not possible to write a constructor such as this:

$$\mathit{SquareNumArray} =_{\text{def}} \lambda\alpha \leq \mathit{Num}.\,\lambda n{:}\alpha.\,\mathit{Array}_{\alpha}(n, n)$$

But this can be written in λ_{Power}, which adds new complexities.

Here is a hint of what the problem is. In Section 4.5.3 on page 112 I mentioned that in a type system with bounded type variables, it can be difficult to tell if a type is the type of a function. Without bounded type variables, it is enough to see if the type is (or reduces to) a Π-type. With bounded type variables, so $\alpha \leq \Pi x{:}A.\,B$, one has to examine the bound to see if it is a function type. In general, this involves iteration along the context and (for higher-order subtyping) normalization steps. This makes type-checking algorithms and meta-theoretical analysis more intricate than without bounded variables. Since λ_{Power} places no restrictions on where bounded variables are allowed, the meta-theoretical studies of $F^{\omega}_{\leq}$ and $\lambda P_{\leq}$ cannot be straightforwardly adapted.

Recently a variant of $F^{\omega}_{\leq}$ with bounded type abstraction was successfully tackled by Compagnoni and Goguen [1997]; this work was mentioned already in Section 4.4.1. Their new technique might be useful for λ_{Power}, but I haven't investigated yet.

Pfenning. Pfenning's *refinement types* extension of LF [Pfenning, 1993] is related to λ_{Power}; it too is a predicative system of dependent types with a form of subtyping. Instead of a relation of subtyping on all types, Pfenning introduces a special class of types (called "sorts") which *refine* the "proper types" of LF. The subtyping relation is only considered between sorts. This is powerful enough to allow some useful examples and yet allows a straightforward proof of conservativity over LF. But subtyping is more restricted than in λ_{Power}; one cannot λ-abstract over a subtype, for example.

There is an analogy between the rough types of λ_{Power} and Pfenning's separation of "proper types" from sorts. Neither proper types nor rough types admit subtyping. Sorts have a subtyping relation, and each sort is a refinement of a designated proper type:

$$\Gamma \vdash R << S \qquad\qquad \Gamma \vdash R, S :: A$$

In λ_{Power}, types have a subtyping relation, and related types have the same rough type:

$$\Gamma \rhd R \leq S \qquad\qquad \Gamma \blacktriangleright R, S \implies P(\tau)$$

The difference is that we have two formal systems rather than one, and rough types have a simpler (non-dependent) structure than refinement types. The dea motivating rough types is the structure of the semantic domains.

Richer types for Z. The types of the specification language Z, in which terms denote sets, are similar to the rough types of λ_{Power}; Z has types of the form κ, $P(\tau)$ and $\tau \times \upsilon$. The official type system of Z has been extended by Spivey [1996] with mechanisms for defining parameterised type constructors called "abbreviations" $a[\vec{\alpha}]$. There is a subtyping relation which begins by taking $a[\vec{\upsilon}] \leq \tau[\vec{\upsilon}/\vec{\alpha}]$, where $a[\vec{\alpha}]$ abbreviates the type τ. Type constructors are always monotonic with respect to the inclusion relation.

The analogy with λ_{Power} is again that a term may have many "richer" types, but only one "official" type, and there is a subtyping relation on richer types. Spivey has considered the algorithmic aspects of type-checking with richer types. However, the richer type system is still non-dependent, so less expressive than LF or λ_{Power}.

4.9.2 *A simplified version of* λ_{Power}

Much of the complexity of λ_{Power} comes from the *contravariant rule* for Π:

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

Apart from PERs assigned to atomic types, this rule is the only way that interesting PERs appear in the semantics of λ_{Power}— considering a PER "interesting" if it does not degenerate to a predicate.

An alternative to Π is the more restricted *equal domains rule*,

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

This rule was originally used in ASL+, and is also used in most (if not all) of the studies of $F^{\omega}_{\leq}$. One reason to avoid the contravariant rule in $F^{\omega}_{\leq}$ is that when used for bounded quantifiers and combined with top types, it renders subtyping relation undecidable [Pierce, 1994].

For $\lambda P_{\leq}$, Compagnoni and I proved that the contravariant rule does not compromise decidability in a system with dependent types but no top type [Aspinall and Compagnoni, 1996]. I conjecture that likewise, the contravariant rule does not cause undecidability of λ_{Power}. However, it is the reason for a more complicated semantics and some proofs are made more difficult. My investigations suggest that with (Π-EQ) in place of (Π), Conjectures 4.11–Conjecture 4.13 are provable using syntactic techniques. Some details concerning η-reduction remain to be checked.

The simplified version of λ_{Power} with an equal domains rule might be adequate in the practical application of ASL+. The cost is needing an extra "trivial" development step occasionally, namely, using a constructor to trim the domain of a parameterised program or specification. An example was shown in Section 2.2.1 on page 38.

4.10 Summary

This chapter described the type system λ_{Power}, a fragment of Cardelli's original power type system [Cardelli, 1988].

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 $Power(A)$ as both a term and a type leads to complication of the meta-theory.

I introduced and studied the syntax of λ_{Power}, given in Figure 4.1 on page 88 and Figure 4.2 following, and the semantics, in Sections 4.6.1–4.6.3.

The semantics is set-based, but uses partial equivalence relations to interpret the equality judgement. The subtyping relation induced by power types is understood as inclusion between PERs. In contrast to other work on semantics of dependent types, the intended model is made by "carving out" from a classical set-hierarchy. Every term in λ_{Power} has a *rough type* which is either an atomic type, or one of the forms $\tau \Rightarrow \upsilon$ or $P(\tau)$, 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 in this chapter are the basic properties of λ_{Power} established in Section 4.4, particularly type correctness, Proposition 4.8; the agreement with the rough typing system, Theorem 4.19; the soundness property for the semantics, Theorem 4.38; and the strong normalization result Corollary 4.46.

5 ASL+

The typed λ-calculus which underlies ASL+ is introduced. The calculus is called $\lambda_{\mathsf{ASL+}}$, and it is obtained by mixing the languages $\lambda_{\leq\{\}}$ and λ_{Power} of the preceding chapters with abstract operators for building programs and specifications. The rough type-checking system and semantics for $\lambda_{\mathsf{ASL+}}$ are based on the definitions given for λ_{Power}.

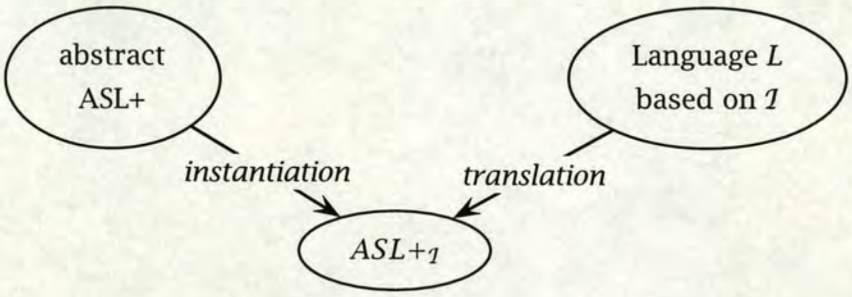
The language ASL+ itself is an instance of $\lambda_{\mathsf{ASL+}}$, given by choosing particular sets of program and specification building operators. The specification building operators are those of ASL.

This new definition improves on the original suggestions for ASL+. Unfortunately, it cannot be used directly for the examples of Chapter 2, because extra translations are needed to handle sharing and dot notation. This drawback is investigated in the latter part of the chapter. Chapters 6 and 7 then introduce a version of ASL+ for a particular institution, which can be used for the Chapter 2 examples.

5.1 *The ASL+Scheme*

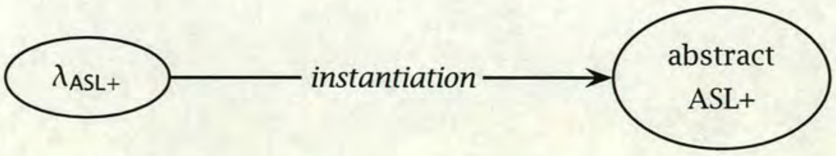
ONE OF THE IDEAS behind ASL+ is that it is *institution independent*. Starting from an institution whose algebras are models of programs in a "core-level" language, we get a modular programming and specification language for free by adding the ASL+ typed λ-calculus on top. In this chapter I shall describe this construction scheme in detail, improving and extending the original proposal of Sannella et al. [1992].

To use ASL+ in a real situation, there is a two-stage process:



A modular programming and specification language L can be given a semantics using ASL+ as a kernel language. But ASL+ itself is an abstract language; we must first *instantiate* it to a particular institution $\mathcal{I}$, upon which the semantics of the core-language of L is based. Instantiating to $\mathcal{I}$ means not only selecting the institution $\mathcal{I}$, but also equipping it with syntactic support: a proof system for proving semantic entailment, and abstract syntax and type-checking systems for representing signatures, models, etc. Since L is a high-level language, we must *translate* the phrases of L into terms of the instantiated kernel language $ASL+_{\mathcal{I}}$. Translating may involve complex context-sensitive handling of parameters, dot notation, etc. as well as simply expanding macros for syntactic sugar.

The abstract formulation of ASL+ described in this chapter is based upon a type system $\lambda_{\mathrm{ASL+}}$, extending the type systems $\lambda_{\leq\{\}}$ and λ_{Power} given previously. In fact there is another step of instantiation to get to ASL+:



The type system $\lambda_{\mathrm{ASL+}}$ has sets of constants which represent operators for building programs and specifications; these constants give the abstract syntax of the core-level programming language and specification language. In the type system, the choice of these constants is not fixed, so the specification language isn't fixed to be ASL. This formalizes the observation by

Sannella et al. [1992] that the choice of the underlying specification language is orthogonal to the design of the rest of the language.[1]

We get ASL+ from $\lambda_{\mathsf{ASL+}}$ by choosing particular sets of constants, based on operations on models in an arbitrary institution. As far as ASL+ is concerned, this reduces the syntax of the underlying languages to something very simple, instead of, for example, beginning from definitions of syntactic representations for the parts of an arbitrary institution (which would be needed to fully specify the syntax of ASL, as explained in Section 1.2). The treatment of the underlying type-checking and proof systems is also encompassed in a simple way, by assuming that the specification and program building operators are simply typed, and by assuming a consequence relation for proving the implementation relations for core-level programs and specifications.

This simple treatment is appealing but it is *too* simple. My goal is to design a language which can directly express the parameterisation mechanisms used in Chapter 2, without needing complex translations. The version of ASL+ given in this chapter does not achieve this goal, for two reasons. First, the language has no built-in notion of *context* for declaring a current environment of modules, nor a built-in way of understanding the dot notation which refers to the context. A context-sensitive translation is needed to fix this, which means duplicating the type-checking rules. Second, the rough type-checking system fails to explain the putting-together-of-signatures during parameter passing necessary to express *sharing* between types, and necessary to give an effective type-checker and an institutional semantics for the language. Again, a complex context-sensitive translation from the source language would be needed to fix this (similar to translations hinted at for Extended ML by Sannella and Tarlecki [1989]). It might be better to begin with a language closer to the goal.

This chapter is structured as follows. The definition of the type system $\lambda_{\mathsf{ASL+}}$ begins in Section 5.2 with the syntax and rough typing system. A model definition for $\lambda_{\mathsf{ASL+}}$ is given in Section 5.3; as for λ_{Power}, an interpretation is defined over roughly-typable terms. The language ASL+ itself is described in Section 5.4, by choosing sets of constants for $\lambda_{\mathsf{ASL+}}$ based on an arbitrary underlying institution $\mathcal{I}$. Section 5.4 also defines the intended model for ASL+ in $\mathcal{I}$. In Section 5.5, the satisfaction system of $\lambda_{\mathsf{ASL+}}$ is defined, which extends the type system of λ_{Power} with rules for singleton types and with a consequence relation $\models^{sat}$ for reasoning about the core-level language. In Section 5.6, some rules are suggested for defining $\models^{sat}$ for ASL+. In Section 5.7, I

[1]The language can be any *ASL-like* language. The type system $\lambda_{\mathsf{ASL+}}$ has some features which are specially for the application to ASL+, which is why it does not have a generic name such as $\lambda_{Power\ \{\}\vdash}$ — thank goodness!

revisit the original proposal for ASL+ to describe reasons behind the revised definitions, and take stock of my achievements.

In Section 5.8 the problems mentioned above are described in more detail. The remainder of the chapter, introduced in Section 5.8.2, considers sharing constraints and putting together signatures. The chapter concludes with mention of some related work in Section 5.11 and a summary in Section 5.12.

5.2 The System $\lambda_{\mathsf{ASL+}}$

This section introduces the type system underlying ASL+, called $\lambda_{\mathsf{ASL+}}$. We get $\lambda_{\mathsf{ASL+}}$ by combining the languages and type systems of $\lambda_{\leq\{\}}$ and λ_{Power}, and adding combinators for building simple programs and specifications. Simple programs and specifications are provided with a consequence relation for proving satisfaction, equality and refinement.

The original language described by Sannella et al. [1992] also has a particular variety of *bounded intersection type*, used to construct specification unions. For simplicity's sake I shall not treat specification union in full generality here, but restrict it to the level of simple specifications as in an earlier formulation of ASL+ [Sannella et al., 1990]. Although there are some questions, one could extend the definitions below to include intersection types following other studies [Pierce, 1991, Compagnoni, 1995].

As usual, the syntax begins with a context-free grammar of pre-terms, in Section 5.2.1. In general, we cannot decide whether a pre-term has a "proper" denotation in the semantics. This is because we can parameterise on specifications, for example writing $\lambda X{:}SP.\,M$, so the definedness of a function application depends on whether the argument satisfies the parameter specification. Checking satisfaction is typically undecidable.

In practice, some amount of checking can be performed: this is the job of the *rough typing* system defined in Section 5.2.2, which follows the same ideas as rough typing for λ_{Power}. For $\lambda_{\mathsf{ASL+}}$, we give the rough typing system first. The "full" typing system, known as the *satisfaction system*, follows much later in Section 5.5.

5.2.1 Syntax

The syntax of $\lambda_{\mathsf{ASL+}}$ is based on three disjoint sets:

- a set *Sign* of atomic type constants, ranged over by Σ.
- a set *PBO* of *program building operators*, ranged over by p.

- a set *SBO* of *specification building operators*, ranged over by s.

and also a fixed countable set $\mathbf{V}$ of variables, ranged over by $X, Y, \ldots$.

Each PBO p or SBO s must be provided with a unique *arity*, which is an element of $\mathit{Sign}^\star \times \mathit{Sign}$. We write $\mathsf{Arity}(p)$ to indicate the arity of p. We assume that for any p or s, there is an effective procedure for calculating $\mathsf{Arity}(p)$ or $\mathsf{Arity}(s)$.

The set of *simple programs* is the set of terms built using only PBOs and variables; the set of *simple specifications* is the set of terms built using only SBOs, variables, and atomic types from *Sign*.

The sets *Sign*, *PBO*, *SBO* together with the Arity() function form a notion of signature for $\lambda_{\mathsf{ASL+}}$ itself. Most of the time, we assume some fixed signature. The language ASL+ is defined by giving a particular signature for $\lambda_{\mathsf{ASL+}}$, in Section 5.4.

The set $\mathbf{T}_{\mathit{Sign},\mathit{PBO},\mathit{SBO}}$ of *pre-terms* over *Sign*, *PBO* and *SBO* is defined by the grammar:

$$\begin{array}{rcl} \mathbf{T} & ::= & \mathit{Sign} \;\mid\; \mathit{PBO}[\mathbf{T}\ldots\mathbf{T}] \;\mid\; \mathit{SBO}[\mathbf{T}\ldots\mathbf{T}] \\ & \mid & \mathbf{V} \;\mid\; \lambda\mathbf{V}{:}\mathbf{T}.\,\mathbf{T} \;\mid\; \mathbf{T}\,\mathbf{T} \;\mid\; \Pi\mathbf{V}{:}\mathbf{T}.\,\mathbf{T} \\ & \mid & \mathit{Spec}(\mathbf{T}) \;\mid\; \{\mathbf{T}\}_{\mathbf{T}} \end{array}$$

We omit the subscript from $\mathbf{T}$. The usual conventions apply for bound variables and the identification of α-equivalent terms. The term $\mathit{Power}(A)$ is a synonym for $\mathit{Spec}(A)$.

The set $C_{\mathit{Sign},\mathit{PBO},\mathit{SBO}}$ of *pre-contexts* consists of sequences of declarations defined via the grammar:

$$C \;::=\; \langle\rangle \;\mid\; C, \mathbf{V} : \mathbf{T}$$

Pre-contexts are ranged over by Γ and we make the usual assumption that no variable is declared more than once in a pre-context.

Setting $\mathcal{K} = \mathit{Sign}$, this syntax properly extends the syntax of both $\lambda_{\leq\{\}}$ (given in Section 3.3) and λ_{Power} (Section 4.2). Compared with the syntax for $\lambda_{\leq\{\}}$, there is now a single syntactic category of terms, and we no longer assume any subtyping relation on the atomic types.

5.2.2 Rough type-checking

Rough type-checking for λ_{Power} was described in Section 4.5. The definitions are similar for $\lambda_{\mathsf{ASL+}}$.

The set $\mathbf{Ty}_{\mathit{Sign}}$ of *rough types of* $\lambda_{\mathsf{ASL+}}$ is defined by the grammar:

$$\frac{\blacktriangleright \Gamma \qquad \mathsf{Arity}(p) = (\Sigma_1 \cdots \Sigma_k, \Sigma) \qquad \text{for } 1 \le i \le k, \Gamma \blacktriangleright M_i \implies \Sigma_i}{\Gamma \blacktriangleright p[M_1 \cdots M_k] \implies \Sigma}$$

$$\frac{\blacktriangleright \Gamma \qquad \mathsf{Arity}(s) = (\Sigma_1 \cdots \Sigma_k, \Sigma) \qquad \text{for } 1 \le i \le k, \Gamma \blacktriangleright A_i \implies P(\Sigma_i)}{\Gamma \blacktriangleright s[A_1 \cdots A_k] \implies P(\Sigma)}$$

$$\frac{\Gamma \blacktriangleright M \implies \tau \qquad \Gamma \blacktriangleright A \implies P(\tau)}{\Gamma \blacktriangleright \{M\}_A \implies P(\tau)}$$

Figure 5.1: Rough typing for $\lambda_{\mathsf{ASL+}}$

$$Ty \ ::= \ Sign \ | \ Ty \Rightarrow Ty \ | \ P(Ty)$$

This is just the same as for λ_{Power}, setting $\mathcal{K} = Sign$.

The rough typing rules for pre-terms of $\lambda_{\mathsf{ASL+}}$ extend those for λ_{Power} shown in Figure 4.3 on page 109, by the new rules shown in Figure 5.1.

The properties of rough typing in Propositions 4.15–4.18 hold for $\lambda_{\mathsf{ASL+}}$ too; the proofs are easily extended. For decidability of rough typing, Proposition 4.15(3), we use the assumption that Arity() is effectively computable. (We also tacitly assume that the identity of elements in *Sign*, *PBO* and *SBO* is decidable.)

Finally, the strong normalization proof of $\beta\eta$-reduction based on rough typing for λ_{Power} given in Section 4.8 extends without difficulty to $\lambda_{\mathsf{ASL+}}$.

5.3 Semantics

The semantics of $\lambda_{\mathsf{ASL+}}$ extends the semantics for λ_{Power} given in Section 4.6. We modify applicative structures to interpret the extra constants in the sets *PBO*, *SBO*, and we add extra axioms to the model definition.

Definition 5.1 ($\lambda_{\mathsf{ASL+}}$ applicative structure).
A $\lambda_{\mathsf{ASL+}}$ *applicative structure* for $(Sign, PBO, SBO)$ is a λ_{Power} applicative structure[2] for the atomic types *Sign*, with the *Const* map also defined on PBOs p and SBOs s, so that:

- $Const(\Sigma) \in \mathcal{D}^{P(\Sigma)}$

[2] see Definition 4.20 on page 117.

- $Const(p) \in \mathcal{D}^{\Sigma_1} \times \cdots \times \mathcal{D}^{\Sigma_k} \to \mathcal{D}^{\Sigma}$
- $Const(s) \in \mathcal{D}^{P(\Sigma_1)} \times \cdots \times \mathcal{D}^{P(\Sigma_k)} \to \mathcal{D}^{P(\Sigma)}$

where p and s have arity $(\Sigma_1 \ldots \Sigma_k, \Sigma)$.

Interpretations and environments are defined just as for λ_{Power}. To give the model definition we need a new construction on relations. Given $R \in \mathrm{REL}(\mathcal{D}^{\tau})$ and $a \in \mathcal{D}^{\tau}$ define

- $[a]_R \in \mathrm{REL}(\mathcal{D}^{\tau})$ by:

$$b\ [a]_R\ c \quad \text{iff} \quad b\ R\ c \ \text{ and } \ b\ R\ a.$$

Fact 5.2. If $R \in \mathrm{PER}(\mathcal{D}^{\tau})$ and $a \in \mathcal{D}^{\tau}$, then $[a]_R \in \mathrm{PER}(\mathcal{D}^{\tau})$.

Definition 5.3 ($\lambda_{\mathsf{ASL+}}$ environment model).
A $\lambda_{\mathsf{ASL+}}$ *environment model* for a $\lambda_{\mathsf{ASL+}}$ structure $\mathcal{D}$ is a λ_{Power} model[3] for $\mathcal{D}$ which satisfies the following axioms for constants, together with the remaining axioms for λ_{Power} models, for all roughly typable contexts Γ and all $\eta \in \mathcal{D}^{\Gamma}$,

CONST $\quad [\![\Sigma]\!]_{\eta} = Const(\Sigma).$

CONST2 $\quad [\![p[M_1 \cdots M_k]]\!]_{\eta} = Const(p)([\![M_1]\!]_{\eta}, \ldots, [\![M_k]\!]_{\eta}).$

CONST3 $\quad [\![s[A_1 \cdots A_k]]\!]_{\eta} = Const(s)([\![A_1]\!]_{\eta}, \ldots, [\![A_k]\!]_{\eta}).$

We assume that $\Gamma \blacktriangleright p[M_1 \cdots M_k] \Rightarrow \Sigma$ and $\Gamma \blacktriangleright s[A_1 \cdots A_k] \Rightarrow P(\Sigma)$, for some signature Σ.
Moreover an $\lambda_{\mathsf{ASL+}}$ model must satisfy the axioms:

SINGLE $\quad R_{[\![\{M\}_A]\!]_{\eta}} = [[\![M]\!]_{\eta}]_{R_{[\![A]\!]_{\eta}}}.$

TOP $\quad R_{[\![\Sigma]\!]_{\eta}} = \mathcal{D}^{\Sigma} \times \mathcal{D}^{\Sigma}.$ □

The constant axioms for signatures, PBOs, and SBOs ensure that their interpretation is given by the *Const* map of the applicative structure. There is no requirement that PBOs respect equivalence classes or that SBOs map PERs to PERs; these conditions are enforced only for particular uses of PBOs and SBOs, later on in Section 5.5.4.

The axiom SINGLE fixes the extension of the interpretation of the singleton type as the equivalence class of $[\![M]\!]$ in the relation which interprets the

[3] see Definition 4.27 on page 121.

type A. The axiom TOP fixes the extension of the interpretation of a signature Σ to be the total relation on $\mathcal{D}^{\Sigma}$.

Coming to model instances, the novel aspect of this semantics is that it allows Σ-specifications SP to be interpreted as *relations* on Σ-algebras, rather than as *sets* (or classes) of Σ-algebras. The reason that TOP fixes the interpretation of atomic types $\Sigma \in \mathit{Sign}$ as the total relation on $\mathcal{D}^{\Sigma}$ is to make the type Σ behave as a "top type" in the inclusion ordering of Σ-specifications. All terms denoting values in $\mathcal{D}^{\Sigma}$ will have type Σ and are considered equal at that type. A term SP denoting a value $[\![SP]\!] \in \mathcal{D}^{P(\Sigma)}$ will be a subtype of Σ, so it will be sound to derive $SP : \mathit{Spec}(\Sigma)$. The reason we want this is so that $\lambda X{:}\,\mathit{Spec}(\Sigma).\, M$ denotes a function parameterised on Σ-specifications, as expected. To be applicable to any Σ-specification, Σ must denote the total relation on Σ-algebras.

5.4 The Language ASL+

The language ASL+ is given by a particular signature for $\lambda_{\mathsf{ASL+}}$, in other words, a choice of *Sign*, *PBO* and *SBO*. In this section I define ASL+ together with its intended model, which is a model of $\lambda_{\mathsf{ASL+}}$.

We begin from an arbitrary institution $\mathcal{I} = \langle \mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models \rangle$. The syntax of ASL+ is based on signatures of the institution as atomic types, so $\mathit{Sign} = |\mathbf{Sign}|$.

The sets *PBO* and *SBO* are chosen to be institution-independent program and specification building operators, and are described in Sections 5.4.3 and Sections 5.4.2 respectively. First we define the underlying λ_{Power} applicative structure.

5.4.1 Applicative structure

We assume that for each $\Sigma \in |\mathbf{Sign}|$, $\mathbf{Mod}(\Sigma)$ is a small category, hence $|\mathbf{Mod}(\Sigma)|$ is a set. This restriction requires that we limit consideration to institutions where algebras over each signature are built within some fixed universe. A more sophisticated semantics might use a category-theoretic model definition for λ_{Power} and ASL+, which would push the problem of size elsewhere.

Now we define a λ_{Power} applicative structure, named $\mathcal{I}$ after the underlying institution:

$$
\begin{aligned}
\mathcal{I}^{\Sigma} &= |\mathbf{Mod}(\Sigma)| \cup \{\diamond\} \\
\mathcal{I}^{\tau \Rightarrow \upsilon} &= (\mathcal{I}^{\tau} \rightharpoonup \mathcal{I}^{\upsilon}) \cup \{\diamond\} \\
\mathcal{I}^{P(\tau)} &= \mathrm{REL}(\mathcal{I}^{\tau}) \cup \{\diamond\} \\
\mathsf{App}(f, m) &= \begin{cases} \diamond & \text{if } f = \diamond, m = \diamond \text{ or } f(m) \text{ is undefined} \\ f(m) & \text{otherwise} \end{cases} \\
\mathsf{Const}(\Sigma) &= (|\mathbf{Mod}(\Sigma)| \times |\mathbf{Mod}(\Sigma)|) \cup \{\diamond\}
\end{aligned}
$$

This is an instance of the full hierarchy partial function structure given in Example 4.23.

This λ_{Power} applicative structure is extended to a $\lambda_{\mathsf{ASL+}}$ applicative structure by defining the sets *SBO* and *PBO* and their interpretations, in the next two sub-sections.

5.4.2 Specification building operators

The specification building operators of ASL+ are those of ASL:

$$\mathit{SBO} =_{\mathrm{def}} \{ I_{\Phi},\ T_{\sigma},\ D_{\sigma},\ M_{\sigma},\ A_{\Phi',\sigma},\ U_{\Sigma} \}$$

Each of these tokens stands for a family of operators, variously indexed by a signature Σ, a set of Σ-sentences Φ, a signature morphism $\sigma : \Sigma \to \Sigma'$, and a set of Σ'-sentences Φ'. The single letters are abbreviations for the operators that were introduced in Section 1.2:

$I_{\Phi}[SP]$	**impose** Φ **on** SP
$T_{\sigma}[SP]$	**translate** SP **by** σ
$D_{\sigma}[SP]$	**derive from** SP **by** σ
$U_{\Sigma}[SP_1, SP_2]$	SP_1 **union** SP_2
$M_{\sigma}[SP]$	**minimal** SP **wrt** σ
$A_{\Phi',\sigma}[SP]$	**abstract** SP **wrt** Φ' **via** σ

(I'm leaving out **iso-close** to make the list a wee bit shorter).

Since we are in an arbitrary institution, there is no concrete syntax for describing signatures Σ, etc.

The interpretation of these SBOs in $\mathcal{I}$ is given in the table below. All the operators are total and, by convention, strict (if any of the argument relations are $\diamond$, then the result is $\diamond$).

	Arity	Interpretation
I_Φ	(Σ,Σ)	$\mathit{Const}(I_\Phi)(R) = \left\{ (m,m') \in R \;\middle|\; m \models^{\mathcal{I}}_\Sigma \Phi \wedge m' \models^{\mathcal{I}}_\Sigma \Phi \right\}$
T_σ	(Σ,Σ')	$\mathit{Const}(T_\sigma)(R) = \{ (m,m') \mid (m\|_\sigma, m'\|_\sigma) \in R \}$
D_σ	(Σ',Σ)	$\mathit{Const}(D_\sigma)(R) = \{ (m\|_\sigma, m'\|_\sigma) \mid (m,m') \in R \}$
M_σ	(Σ',Σ')	$\mathit{Const}(M_\sigma)(R) = \left\{ (m,m') \in R \;\middle|\; m,m' \ \sigma\text{-minimal in } \|\mathbf{Mod}^{\mathcal{I}}(\Sigma)\| \right\}$
$A_{\Phi',\sigma}$	(Σ,Σ)	$\mathit{Const}(A_{\Phi',\sigma})(R) = \left\{ (m,m') \;\middle|\; \begin{array}{l} m \equiv^\sigma_{\Phi'} m' \text{ and } \exists m''. \\ (m,m'') \in R \vee (m'',m) \in R \end{array} \right\}$
U_Σ	$(\Sigma\,\Sigma,\Sigma)$	$\mathit{Const}(U_\Sigma)(R_1,R_2) = \{ (m,m') \mid (m,m') \in R_1 \wedge (m,m') \in R_2 \}$

This generalises the ASL semantics given in Definition 1.6 on page 14; see there for definitions used above. Mostly, the usual semantics is generalised by taking the product of the operator with itself. The **abstract** operator is slightly different: it returns the parts of the equivalence relation $\equiv^\sigma_\Phi$ which have some intersection with the argument relation. This only pays attention to the domain of the argument relation, and always returns a "full" subrelation of $\equiv^\sigma_\Phi$. The idea behind this definition is explained in Section 5.4.5.

5.4.3 *Program building operators*

Program building operators for an arbitrary institution have been less well studied than specification building operators. The following examples are given as operations for manipulating constructors by Sannella and Tarlecki [1988b]; some of these rely on additional assumptions about the underlying institution.

$$\mathit{PBO} =_{\text{def}} \{ R_\sigma, S_{\sigma'}, Q_\Phi, E_{\Phi',\sigma}, U_{\sigma_1,\sigma_2} \}$$

These operators are indexed by $\sigma : \Sigma \to \Sigma'$, $\sigma' : \Sigma' \to \Sigma$, $\Phi \subseteq \mathbf{Sen}(\Sigma)$, $\Phi' \subseteq \mathbf{Sen}(\Sigma')$, $\sigma_1 : \Sigma \to \Sigma_1$, and $\sigma_2 : \Sigma \to \Sigma_2$. Descriptive names for the operations are:

$R_\sigma[P]$	**reduct of** P **by** σ
$S_{\sigma'}[P]$	**restrict** P **via** σ'
$Q_\Phi[P]$	**quotient** P by Φ
$E_{\Phi',\sigma}[P]$	**extend** P to Φ' **via** σ
$U_{\sigma_1,\sigma_2}[P_1, P_2]$	**a-union** P_1, σ_1 **and** P_2, σ_2

Here is a brief explanation of them.

reduct of P **by** σ is the reduct operation on models. It can be used to hide some parts of a program.

restrict P **via** σ' restricts P to a minimal expansion of its σ' reduct, in other words, its unique σ'-minimal subalgebra. (This assumes that the institution is supplied with a notion of subalgebra, and unique restrictions exist; see Section 1.2.) This can be used to remove "junk" from models.

quotient P **by** Φ returns the quotient of P with respect to the equivalence relation on sorts induced by the set of equations Φ. (This assumes several things about the underlying institution.) This can be used to make new datatypes by quotienting old ones.

extend P **to** Φ' **via** σ yields the unique free extension of P to a model of Φ', where Φ' is a set of equations over Σ'. This relies on the existence of free functors $F_{\sigma,\Phi'} : \mathbf{Mod}(\Sigma) \to [\![\Phi']\!]$ in the institution. This allows definition of freely-generated datatypes, for example.

a-union P_1, σ_1 **and** P_2, σ_2 returns the amalgamation of P_1 and P_2 with respect to σ_1, σ_2. This assumes that the institution has pushouts in the category of signatures, and that these give rise to unique amalgamations in the model categories. When defined, the result is a Σ^*-model, where

$$\begin{tikzcd}
\Sigma_1 \arrow[r, dashed, "{\sigma_2}^*"] & \Sigma^* \\
\Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 \arrow[u, dashed, "{\sigma_1}^*"']
\end{tikzcd}$$

is a pushout in **Sign**. This operation allows two programs to be combined, provided they have the same implementation of shared symbols.

The semantics is defined on models rather than isomorphism classes, so some of the operators must select canonical representatives. In practice this is not a problem, because we work with finite syntactic representations of models and operations on them.

The semantics of the PBOs is given in the table below. Again, each operator is strict: if any of the arguments is $\diamond$, then so is the result. All operators are total except for the **a-union** operator; however, other operators may only be partial, depending on the institution or design choice. (For example, whilst $F_{\sigma,\Phi'}$ exists in some institutions, it is not guaranteed to result in a model which satisfies Φ' exactly; in these cases we might choose to make **extend** P to Φ' **via** σ undefined).

	Arity	Interpretation
R_σ	(Σ',Σ)	$\mathit{Const}(R_\sigma)(m) = m\vert_\sigma$
$S_{\sigma'}$	(Σ,Σ)	$\mathit{Const}(S_{\sigma'})(m) = \sigma'$-subalgebra of m
Q_Φ	(Σ,Σ)	$\mathit{Const}(Q_\Phi)(m) = m/\equiv_\Phi$
$E_{\Phi',\sigma}$	(Σ,Σ')	$\mathit{Const}(E_{\Phi',\sigma})(m) = F_{\sigma,\Phi'}(m)$
U_{σ_1,σ_2}	$(\Sigma_1\ \Sigma_2,\Sigma')$	$\mathit{Const}(U_{\sigma_1,\sigma_2})(m_1,m_2) = \begin{cases} \text{amalgamation of } m_1, m_2 & \text{if } m_1\vert_{\sigma_1} = m_2\vert_{\sigma_2}, \\ \diamond & \text{otherwise.} \end{cases}$

For more details of these operations, see Sannella and Tarlecki [1988b].

5.4.4 *Interpretation in $\mathcal{I}$*

The previous sections defined a $\lambda_{\mathsf{ASL+}}$ applicative structure. Now we can define a meaning function for this structure.

Definition 5.4. Using the definitions of the *Const* map given above, we define an interpretation in the applicative structure $\mathcal{I}$ given in Section 5.4.1 by:

$$\begin{aligned}
\mathit{Rel}(A) &= A \\
[\![\Gamma \blacktriangleright x \Rightarrow \tau\ ]\!]_\eta &= \eta^\eta(x) \\
[\![\Gamma \blacktriangleright \Sigma \Rightarrow P(\tau)]\!]_\eta &= |\mathbf{Mod}(\Sigma)| \times |\mathbf{Mod}(\Sigma)| \\
\Gamma \blacktriangleright p[M_1 \cdots M_k] \Rightarrow \Sigma &= \mathit{Const}(p)([\![M_1]\!]_\eta \ldots [\![M_k]\!]_\eta) \\
\Gamma \blacktriangleright s[A_1 \cdots A_k] \Rightarrow P(\Sigma) &= \mathit{Const}(s)([\![A_1]\!]_\eta \ldots [\![A_k]\!]_\eta) \\
\ldots
\end{aligned}$$

where the rest of the interpretation is defined in the same way as the interpretation given in Example 4.28 on page 122. □

Lemma 5.5. *The interpretation defined in Definition 5.4 yields a model of* $\lambda_{\mathsf{ASL+}}$ *for* $(\mathit{Sign}, \mathit{SBO}, \mathit{PBO})$.

Proof As the proof of Lemma 4.29 on page 122, proving the extra axioms in Definition 5.3. □

5.4.5 Motivating specifications-as-PERs

Section 5.4.2 generalised the semantics of the ASL specification building operators from operations on sets (or classes) to operations on PERs. Why use PERs for specifications?

For types, the reason to generalise from types-as-sets to types-as-PERs was explained in Section 4.6 on page 116. With subtyping, two terms can be equal at one type, but different at another (examples were in Sections 2.2.6 and 3.1), so the interpretation of types must include an equality relation as well as a collection of elements.

For specifications, a semantics for ASL+ would not need to use PERs throughout. Instead it could use the usual semantics for SBOs and introduce PERs only at higher-types. But using PERs throughout offers an interesting new way of treating observational equivalence.

The **abstract** operator of ASL closes up a class of models under a relation of *observational equivalence*, as described in Section 1.2. Usually, **abstract** *SP* **wrt** Φ **via** σ is interpreted as:

$$\left\{ m \;\middle|\; m \in \mathbf{Mod}(\Sigma) \bigwedge \exists m' \in [\![SP]\!].\, m \equiv^{\Phi}_{\sigma} m' \right\}$$

In the PER semantics a specification is interpreted as a PER, which gives the collection of models which satisfy it, *together with an equivalence relation on those models.* The equivalence relation is intended to be an appropriate observational equivalence. The **abstract** operator now retains equivalence classes in the result:

$$\left\{ (m, m') \;\middle|\; m \in \mathbf{Mod}(\Sigma) \bigwedge \exists m' \in \mathit{Dom}([\![SP]\!]).\, m \equiv^{\Phi}_{\sigma} m' \right\}$$

The equivalence relation of the argument *SP* is replaced with a different relation, only paying attention to the models of *SP*. This can either hide or reveal equivalences. Just as in ASL, the use of observational equivalence is controlled by the low-level **abstract** operator; a translation from a high-level language would insert **abstract** automatically in appropriate places.

Here is the interesting point. The equivalence relation is respected everywhere in the semantics. In particular, the interpretation of a Π-specification $\Pi X{:}SP.\, SP'$ is the collection of functions which map implementations of *SP* to implementations of *SP'*, *preserving equivalence.* When the equivalence relation is observational equivalence, then observationally equivalent models of *SP* must be mapped to observationally equivalent models of *SP'*. This is precisely the requirement of *stability*, which should be met by all definable constructors in a decent programming language, researchers say [Schoett, 1987, Sannella and Tarlecki, 1997]. Stability relates to providing abstraction across module boundaries; here it is built into the semantics of parameterisation.[4]

This needs further research. Alternative generalisations of the operators should be investigated, and we should study refinement at higher types to see whether the type-theoretical rules and semantics here give useful meanings of higher-order observational equivalence.

5.5 The Satisfaction System of $\lambda_{\mathsf{ASL+}}$

The *satisfaction system* of $\lambda_{\mathsf{ASL+}}$ is the type system which forms the basis of the system for proving satisfaction in ASL+, which will be explained in Section 5.6. The satisfaction system of $\lambda_{\mathsf{ASL+}}$ extends the type system of λ_{Power} with rules for singletons, similar to those of $\lambda_{\leq\{\}}$, and with rules for using an external proof system for PBOs and SBOs.

Section 5.5.1 defines the form of consequence relation used to capture the external proof system. Section 5.5.2 defines the satisfaction system itself. Section 5.5.3 extends the meta-theory of λ_{Power} to $\lambda_{\mathsf{ASL+}}$ and Section 5.5.4 presents a soundness proof for the satisfaction system.

5.5.1 Consequence relations for $\lambda_{\mathsf{ASL+}}$

To prove satisfaction, equality, and refinement for terms written with PBOs and SBOs, we need to appeal to an externally-provided proof system. To define the type system of $\lambda_{\mathsf{ASL+}}$, we don't need to know the detailed structure of this external proof system; it is enough to know the consequence relation (CR) that it gives rise to. But the CR must be *schematic* in some variables so we can substitute for complex ASL+ terms.

Rather than give a new definition of CR which axiomatises the notion of free variables in sentences, we can simply re-use the signature-indexed definition of CR in Definition 1.3, using contexts Γ as the notion of signature.

[4]This shows an alternative to the suggestion in Sannella et al. [1992, Section 4] that specifying stability would violate the *regularity* condition that every specification of a parameterised algebra can be expressed as a cartesian product. Here, requiring or not requiring stability corresponds to particular uses of **abstract**, and all specifications of parameterised algebras are regular.

Contexts have a categorical structure: morphisms between contexts are substitutions, with renaming and weakening as special cases.

In fact, indexing the CR by an abstract notion of context seems the easiest way to characterise a schematic consequence, compared to indexing by sets of free variables. Without the definition of the syntax of terms and sentences to hand, substitution is difficult to axiomatise abstractly [for remarks, see Avron, 1992].

Definition 1.3 works because to express a structural proof system, we need a single context for the whole consequence, rather than different contexts over each sentence. This is a "truth" type of interpretation, as opposed to a "validity" type. (Both types of interpretation, and richer notions of consequence relation, are described in Harper et al. [1989].) Informally speaking, the single context declares the meta-variables in some composition of rules from the proof system, and the meta-variables may be instantiated throughout for ASL+ terms.

Sentences of a consequence relation for $\lambda_{\mathsf{ASL+}}$ have two forms:

$$s \ ::= \ P = P : SP \quad | \quad SP \leq SP$$

where P ranges over simple programs and SP ranges over simple specifications, described in Section 5.2.1. I will use s to range over sentences, and Δ to range over sets of sentences. The sentence $P : SP$ abbreviates $P = P : SP$. Sentences are roughly typed by the rules:

$$\frac{\Gamma \blacktriangleright P, P' \Rightarrow \Sigma \qquad \Gamma \blacktriangleright SP \Rightarrow \mathsf{P}(\Sigma)}{\Gamma \blacktriangleright P = P' : SP \Rightarrow \Sigma} \qquad\qquad \frac{\Gamma \blacktriangleright SP, SP' \Rightarrow \mathsf{P}(\Sigma)}{\Gamma \blacktriangleright SP \leq SP' \Rightarrow \Sigma}$$

Fully-applied PBOs and SBOs have rough types of the form Σ or $\mathsf{P}(\Sigma)$, so it is enough for rough typing to use contexts Γ whose types have the form Σ or $\mathsf{Spec}(\Sigma)$. This leads to the next definition.

Definition 5.6 (Consequence relation for $\lambda_{\mathsf{ASL+}}$).
A $(\mathsf{Sign}, \mathsf{SBO}, \mathsf{PBO})$ *consequence relation* is a CR $\vdash^{sat}$ where

- $\mathbf{Sign}^{\vdash^{sat}}$ is the category in which
  - an object is a $\lambda_{\mathsf{ASL+}}$ pre-context Γ where each declaration has the form $X : \Sigma$ or $X : \mathsf{Spec}(\Sigma)$, for some $\Sigma \in \mathsf{Sign}$;
  - a morphism $\gamma : \Gamma \to \Gamma'$ is a substitution from $Dom(\Gamma)$ to simple programs and specifications over $\mathsf{Sign}, \mathsf{SBO}, \mathsf{PBO}$, such that

    $$\Gamma \blacktriangleright X \Rightarrow \tau \quad \Rightarrow \quad \Gamma' \blacktriangleright \gamma(X) \Rightarrow \tau$$

    for all $X \in Dom(\Gamma)$.

- $\mathbf{Sen}^{\models^{sat}}(\Gamma) = \{ s \mid \exists \Sigma.\ \Gamma \blacktriangleright s \Rightarrow \Sigma \}$.
  A substitution γ extends to terms in the obvious way; then $\mathbf{Sen}^{\models^{sat}}(\gamma)(s)$ is given by applying γ to the terms of s.
- Each relation $\models^{sat}_{\Gamma}$ meets the following closure conditions:

1. *Symmetry.*	If $\Delta \models^{sat}_{\Gamma} P = P' : SP$ then $\Delta \models^{sat}_{\Gamma} P' = P : SP$.
2. *Transitivity for* $=$.	If $\Delta \models^{sat}_{\Gamma} P_1 = P_2 : SP$ and $\Delta \models^{sat}_{\Gamma} P_2 = P_3 : SP$, then $\Delta \models^{sat}_{\Gamma} P_1 = P_3 : SP$.
3. *Transitivity for* $\leq$.	If $\Delta \models^{sat}_{\Gamma} SP \leq SP'$ and $\Delta \models^{sat}_{\Gamma} SP' \leq SP''$ then $\Delta \models^{sat}_{\Gamma} SP \leq SP''$.
4. *Subsumption.*	If $\Delta \models^{sat}_{\Gamma} P = P' : SP$ and $\Delta \models^{sat}_{\Gamma} SP \leq SP'$ then $\Delta \models^{sat}_{\Gamma} P = P' : SP'$.
5. *Top.*	If $\Delta \models^{sat}_{\Gamma} P : \Sigma$ and $\Delta \models^{sat}_{\Gamma} P' : \Sigma$ then $\Delta \models^{sat}_{\Gamma} P = P' : \Sigma$.
6. *Formation for* $=$.	$\Delta \models^{sat}_{\Gamma} P = P' : SP$ implies $\Delta \models^{sat}_{\Gamma} SP \leq \Sigma$.
7. *Formation for* $\leq$.	Either $\Delta \models^{sat}_{\Gamma} SP \leq SP'$ or $\Delta \models^{sat}_{\Gamma} SP' \leq SP$ implies $\Delta \models^{sat}_{\Gamma} SP \leq SP$ and $\Delta \models^{sat}_{\Gamma} SP \leq \Sigma$. □

Using properties of rough typing, it is easy to show that $\mathbf{Sign}^{\models^{sat}}$ is a category and $\mathbf{Sen}^{\models^{sat}}$ is a functor. The closure conditions on $\models^{sat}$ capture intuitive conditions corresponding to familiar rules from subtyping systems. The atomic types Σ are treated as "top" types: any two Σ-programs are equal at type Σ, and by formation for $\leq$, any Σ-specification is a refinement of Σ. This accords with the model axiom which fixes the interpretation of Σ.

None of the requirements assert a sentence directly; for example, we do not have that $\Gamma \blacktriangleright SP \Rightarrow \Sigma$ implies $\Delta \models^{sat}_{\Gamma} SP \leq \Sigma$. This allows for PBOs or SBOs to be *partial*, requiring some semantic conditions in the antecedent of the consequence relation to be well-defined. A typical example is the amalgamation operator, where the program **a-union** P_1, σ_1 **and** P_2, σ_2 is only well-defined if $[\![P_1]\!]|_{\sigma_1} = [\![P_2]\!]|_{\sigma_2}$. This program can have a rough type yet an undefined denotation, in which case it should be excluded from the satisfaction system.

5.5.2 The satisfaction system

The rules of the satisfaction system for $\lambda_{\mathsf{ASL+}}$ given in Figures 5.2 and 5.3 extend the typing and equality judgements of λ_{Power} given in Figures 4.1 and 4.2 on pages 88 and 89.

Rules for singletons

The singleton type rules are shown in Figure 5.2 on the following page. These are taken from the system $\lambda_{\leq\{\}}$ defined in Chapter 3, modifying for the different judgement forms and for power types.

Here is a quick review, comparing the rules of $\lambda_{\leq\{\}}$ against the new ones. First, we need no formation rule for singletons, because the rule ({}-SUB-TAG) does the job.[5] The typing rules given in Figure 3.2 on page 80 are already in λ_{Power}, as are the equality rules in Figure 3.3 on page 81. From the subtyping rules in Figure 3.4 on page 82, we take (SUB-{}) and (SUB-EQ-ITER), which appear in Figure 5.2 as ({}-SUB-TAG) and ({}-{}-SUB). The "subtyping of equivalence classes" part of the composite rule (SUB-EQ-SYM) appears as ({}-SUB); the symmetry part of (SUB-EQ-SYM) is not needed because we already have symmetry for equality. Finally, we have a rule ({}-ELIM) which connects inhabitation of singleton types with the equality judgement, introducing the equality $\Gamma \rhd M = N : A$ from a singleton typing $\Gamma \rhd M : \{N\}_A$. (The opposite direction of this rule is admissible.)

Rules for PBOs and SBOs

The rules for PBOs and SBOs are shown in Figure 5.3 on page 156.

These rules use substitutions θ to instantiate the consequence relation $\models^{sat}$. This is a richer form of substitution than morphisms in $\mathbf{Sign}^{\models^{sat}}$, because the target context can be any pre-context of $\lambda_{\text{ASL+}}$, and the typing requirement is in the full system. Specifically, we write $\theta : \Gamma' \to \Gamma$ iff

$$\Gamma \rhd \theta(X) \ : \ \theta(\Gamma'(X))$$

for all $X \in Dom(\Gamma')$. So $\theta : \Gamma' \to \Gamma$ is a shorthand for several premises.[6]

The premise $\Gamma \rhd \theta(\Delta)$ is also an abbreviation. When $\Delta = \{ s_1, \ldots, s_n \}$ and is non empty, then $\Gamma \rhd \theta(\Delta)$ stands for the n premises:

$$\Gamma \rhd \theta(s_1), \quad \ldots, \quad \Gamma \rhd \theta(s_n).$$

Again, substitution on a sentence s_i is defined by extending θ to the terms of s_i. If $s_i \equiv SP \leq SP'$, for example, then $\Gamma \rhd \theta(s_i)$ stands for the $\lambda_{\text{ASL+}}$ judgement $\Gamma \rhd \theta(SP) : \mathit{Spec}(\theta(SP'))$.

[5] Remember that in λ_{Power}, $\Gamma \rhd A$ *type* abbreviates $\Gamma \rhd A : \mathit{Power}(B)$ for some B.

[6] In Figure 5.3, the source context Γ' of substitutions θ always comes from an instance of $\models^{sat}_{\Gamma'}$, which means that $\Gamma'(X) \equiv \Sigma$ or $\Gamma'(X) \equiv \mathit{Spec}(\Sigma)$, so the substitution on the right-hand side is superfluous.

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd M : \{M\}_A} \qquad (\{\}\text{-INTRO})$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A : \mathit{Power}(A)} \qquad (\{\}\text{-SUB-TAG})$$

$$\frac{\Gamma \rhd M : A \qquad \Gamma \rhd A : \mathit{Power}(B)}{\Gamma \rhd \{M\}_A : \mathit{Power}(\{M\}_B)} \qquad (\{\}\text{-SUB})$$

$$\frac{\Gamma \rhd M : A}{\Gamma \rhd \{M\}_A : \mathit{Power}(\{M\}_{\{M\}_A})} \qquad (\{\}\text{-}\{\}\text{-SUB})$$

$$\frac{\Gamma \rhd M : \{N\}_A}{\Gamma \rhd M = N : A} \qquad (\{\}\text{-ELIM})$$

Figure 5.2: Singleton types in $\lambda_{\mathsf{ASL+}}$

$$\frac{\Delta \vdash^{sat}_{\Gamma'} P : SP \qquad \theta : \Gamma' \to \Gamma \qquad \Gamma \rhd \theta(\Delta)}{\Gamma \rhd \theta(P) \;:\; \theta(SP)} \quad \text{(PBO)}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma'} SP \leq SP' \qquad \theta : \Gamma' \to \Gamma \qquad \Gamma \rhd \theta(\Delta)}{\Gamma \rhd \theta(SP) \;:\; \mathit{Spec}(\theta(SP'))} \quad \text{(SBO)}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma'} P = P' : SP \qquad \theta : \Gamma' \to \Gamma \qquad \Gamma \rhd \theta(\Delta)}{\Gamma \rhd \theta(P) = \theta(P') \;:\; \theta(SP)} \quad \text{(EQ-PBO)}$$

$$\frac{\begin{array}{l}\Delta \vdash^{sat}_{\Gamma'} SP \leq SP' \\ \Delta \vdash^{sat}_{\Gamma'} SP' \leq SP \qquad \theta : \Gamma' \to \Gamma \qquad \Gamma \rhd \theta(\Delta)\end{array}}{\Gamma \rhd \theta(SP) = \theta(SP') : \mathit{Spec}(\theta(SP'))} \quad \text{(EQ-SBO)}$$

$$\frac{\Gamma \rhd M : \Sigma \qquad \Gamma \rhd N : \Sigma}{\Gamma \rhd M = N : \Sigma} \quad \text{(EQ-TOP)}$$

Figure 5.3: SBOs and PBOS in $\lambda_{\mathsf{ASL+}}$

When $\Delta = \emptyset$, then by convention $\Gamma \rhd \theta(\Delta)$ stands for the single premise $\rhd \Gamma$. This ensures that the context Γ is well-formed in the conclusion of each rule which uses $\vdash^{sat}$.

The rule (PBO) proves that a program term built with simple program constructors satisfies a specification built with simple specification constructors. Recall that $P : SP$ is defined to mean $P = P : SP$ as a sentence of $\vdash^{sat}$. The rule (EQ-PBO) handles the more general assertion, introducing an equality into the system. The rule (SBO) proves refinement between two specifications built with simple constructors, and the equality of such specifications is proved by showing that each is a refinement of the other, with the rule (EQ-SBO).

Substitution is grafted on to each of these rules as $\vdash^{sat}$ is added to the typing and equality judgements. In the combined proof system, we can use the satisfaction rules for the $\lambda_{\mathsf{ASL+}}$ type constructors and the proof system of $\vdash^{sat}$ one after another, as the structure of a specification or program is broken down. Substitution is needed because $\lambda_{\mathsf{ASL+}}$ assumptions can only have the form $X : A$, and not $M : A$ for arbitrary terms M. We think of assumptions $X : A$ as typing declarations for free variables rather than logical assumptions, but the distinction is blurred somewhat at this stage.[7]

The last rule in Figure 5.3, (EQ-TOP), treats an atomic signature type $\Sigma \in \mathit{Sign}$ as a top type, and asserts that two terms which denote Σ-algebras are indistinguishable at type Σ.

5.5.3 Properties of satisfaction

In this section we extend the results of Section 4.4 to $\lambda_{\mathsf{ASL+}}$. First we prove the agreement property between the rough typing system and the satisfaction system.

This is a re-statement of Theorem 4.19 on page 111.

Theorem 5.7 (Agreement of rough typing for $\lambda_{\mathsf{ASL+}}$).

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

Proof We sketch one new case for the second part, for the rule (EQ-PBO). The remaining cases to check are similar or straightforward.

Case (EQ-PBO): By the definition of sentences and proof relations, $\Delta \vdash^{sat}_{\Gamma'} P = P' : SP$ means that the required result holds in Γ' for P, P' and SP. By the translation condition for consequence relations, we can assume that the variables of Γ' and Γ are distinct. Using the extension of Proposition 4.16, we thin the context to Γ', Γ and then use substitution repeatedly to substitute each variable $X \in \mathit{Dom}(\Gamma')$ for the term $\theta(X)$, preserving the rough typing and yielding $\Gamma \blacktriangleright \theta(P) \implies \Sigma$, $\Gamma \blacktriangleright \theta(P') \implies \Sigma$ and $\Gamma \blacktriangleright \theta(SP) \implies \mathsf{P}(\Sigma)$, as required. □

To prove thinning, substitution and bound narrowing, we must simultaneously prove corresponding statements about substitutions θ. These statements are gathered in the lemma below. For the substitution case, we use a substitution $\theta[N/X]$, which is defined by:

$$(\theta[N/X])\,(Y) = \theta(Y)\,[N/X]$$

This corresponds to the substitution of N for X in every part of θ.

[7]Indeed, a system indexed by rough-contexts with arbitrary assumptions might be worth studying, at least because this would be a consequence relation itself, and there would be an easy way to state conservativity over $\vdash^{sat}$.

Lemma 5.8 (Substitutions for roughly typed variables).

1. *If* $\theta : \Gamma' \to \Gamma$ *and* $\Gamma \subseteq \Gamma'$ *with* $\rhd \Gamma'$ *then* $\theta : \Gamma' \to \Gamma'$.
2. *If* $\theta : \Gamma' \to \Gamma, X : A, \Gamma'$ *and* $\Gamma \rhd N : A$, *then* $\theta[N/X] : \Gamma' \to \Gamma, \Gamma'[N/X]$.
3. *If* $\theta : \Gamma' \to \Gamma, X : A, \Gamma'$ *and* $\Gamma \rhd A' : \mathsf{Power}(A)$ *then* $\theta : \Gamma' \to \Gamma, X : A', \Gamma'$.

Proof Simultaneously with the extensions to $\lambda_{\mathsf{ASL+}}$ of Proposition 4.4(3), Proposition 4.5 and Proposition 4.6. In each case, the induction hypothesis for a statement above helps prove the proposition for the rules in Figure 5.3. □

The remainder of Proposition 4.4 extends to $\lambda_{\mathsf{ASL+}}$ easily, so this establishes the extension of Propositions 4.4–4.6. The formation property Proposition 4.7 needs an extra case, and care with the PBO and SBO rules.

Proposition 5.9 (Formation for $\lambda_{\mathsf{ASL+}}$).

1. $\Gamma \rhd \lambda x{:}A.M : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathit{type} \quad \mathit{and} \quad \exists B.\ \ \Gamma, x : A \rhd M : B.$
2. $\Gamma \rhd M\,N : C \quad \Longrightarrow \quad \exists A, B.\ \ \Gamma \rhd M : \Pi x{:}A.B \quad \mathit{and} \quad \Gamma \rhd N : A.$
3. $\Gamma \rhd \Pi x{:}A.B : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathit{type} \quad \mathit{and} \quad \Gamma, x : A \rhd B\ \mathit{type}.$
4. $\Gamma \rhd \mathsf{Power}(A) : C \quad \Longrightarrow \quad \Gamma \rhd A\ \mathit{type}.$
5. $\Gamma \rhd \{M\}_A : C \quad \Longrightarrow \quad \Gamma \rhd M : A.$

Proof As for Proposition 4.7, except that term in question could now be introduced by (PBO) or (SBO). Consider (PBO); either $\theta(P) \equiv p[M_1 \ldots M_n]$, or $\theta(P) \equiv N$ and N may have the term former of interest outermost. If so, it must be that $P \equiv Y$ for some variable Y, with $\theta(Y) = N$. By the definition of $\theta : \Gamma' \to \Gamma$ we have a subderivation of $\Gamma \rhd N : \Gamma'(Y)$. We can use the induction hypothesis for this derivation to get the result. Similarly for (SBO). □

Type correctness extends Proposition 4.8. The statement is the same.

Proposition 5.10 (Type correctness for $\lambda_{\mathsf{ASL+}}$).

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

Proof The cases for the singleton rules are straightforward, making use of Proposition 5.9 for ({}-ELIM). For the PBO and SBO rules, we show the proof for (EQ-PBO).

Case (EQ-PBO): For part 1 we must show $\Gamma \rhd \theta(SP)\ \mathit{type}$. By the reflexivity requirement for $\models^{sat}$, we must have that $\Delta \models^{sat}_{\Gamma} SP \leq SP$, so we can use (SBO) to derive $\Gamma \rhd \theta(SP) : \mathsf{Spec}(\theta(SP))$ as required. For part 2, to show

$\Gamma \rhd \theta(P) : \theta(SP)$, use (PBO); to show $\Gamma \rhd \theta(P') : \theta(SP)$, use (PBO) too, using the symmetry requirement for $\vdash^{sat}$. □

Finally, the admissible rules shown in Propositions 4.9 and 4.10 remain admissible in $\lambda_{\mathsf{ASL+}}$, since their proofs rely only on the preceding propositions and derivations in λ_{Power}.

5.5.4 *Soundness of satisfaction*

In this section we extend to $\lambda_{\mathsf{ASL+}}$ the soundness result for λ_{Power} given in Section 4.7. We must assume that the consequence relation $\vdash^{sat}$ is itself sound.

Let η_1, η_2 be a pair of environments such that $\eta_1 \; [\![\Gamma]\!] \; \eta_2$, where the declarations in Γ only have the form Σ or $\mathit{Spec}(\Sigma)$. We write

$$(\eta_1, \eta_2) \models P = P' : SP \quad \iff \quad [\![P]\!]_{\eta_1} \; R_{[\![SP]\!]_{\eta_1}} \; [\![P']\!]_{\eta_2}$$

and

$$\begin{array}{lcl}(\eta_1, \eta_2) \models SP \leq SP' & \iff & R_{[\![SP]\!]_{\eta_1}} = R_{[\![SP]\!]_{\eta_2}} \in \mathrm{PER}(\mathcal{D}^{\Sigma}) \\ & & \bigwedge R_{[\![SP]\!]_{\eta_1}} \subseteq R_{[\![SP']\!]_{\eta_1}}\end{array}$$

Given Γ-sentences of a $\lambda_{\mathsf{ASL+}}$ consequence relation,

Definition 5.11 (Sound consequence relations).
A $\lambda_{\mathsf{ASL+}}$ consequence relation $\vdash^{sat}$ is *sound* if

$$\{ s_1, \ldots, s_n \} \vdash^{sat}_{\Gamma} s$$

implies that

$$(\eta_1, \eta_2) \models s_1 \; \bigwedge \; \ldots \; \bigwedge \; (\eta_1, \eta_2) \models s_n \quad \Longrightarrow \quad (\eta_1, \eta_2) \models s$$

for all $\eta_1 \; [\![\Gamma]\!] \; \eta_2$.

The interpretation of contexts and conventions for environments are unchanged. The following theorem is a re-statement of Theorem 4.38 on page 126.

Theorem 5.12 (Soundness of satisfaction for $\lambda_{\mathsf{ASL+}}$ models).
Let $\vdash^{sat}$ be a sound consequence relation according to Definition 5.11.

1. *If $\rhd \Gamma$ then $[\![\Gamma]\!] \in \mathrm{PER}(\mathcal{D}^{\Gamma})$*
2. *If $\Gamma \rhd M : A$, then for all $\eta_1, \eta_2 \in \mathcal{D}^{\Gamma}$,*

$$\eta_1 \; [\![\Gamma]\!] \; \eta_2 \quad \Longrightarrow \quad [\![M]\!]_{\eta_1} \; R_{[\![A]\!]_{\eta_1}} \; [\![M]\!]_{\eta_2}.$$

Proof systems for the core-level are not the main topic of this thesis, so the system shown here is incomplete and imperfect. I provide it just to give some ideas of what the rules for ASL+ PBOs and SBOs look like, and to see how they fit with the satisfaction rules of $\lambda_{\mathsf{ASL+}}$.

There are several ways that a relation $\Delta \models^{sat}_{\Gamma} s$ could be presented. A natural-deduction style of presentation might be the clearest, with rules like this:

$$\frac{}{SP \leq SP} \qquad \frac{SP_1 \leq SP_2 \qquad SP_2 \leq SP_3}{SP_1 \leq SP_3}$$

then $\Delta \models^{sat}_{\Gamma} s$ holds if there is derivation of s from assumptions Δ, when s and Δ are well-typed in Γ. This convention hides the context of assumptions and free variables. Another convention is to identify variables in Γ with meta-variables in the presentation, because $\models^{sat}_{\Gamma}$ is supposed to capture the schematic consequences of derived proof rules.

Because I only present a few rules, I shall be overly explicit and not use any conventions; I define the relation $\Delta \models^{sat}_{\Gamma} s$ directly, as the least relation generated inductively by a set of rules. This is an unusually verbose form of presentation for a logic, and it means adding as rules some of the structural properties that the CR must satisfy.[8] Moreover, I shall be careful to include all the typing information for each rule.

The rules are shown in Figures 5.4–5.6. We assume that there is a proof system provided for the logic of the underlying institution $\mathcal{I}$, which gives rise to a deduction relation $\vdash^{\mathcal{I}}$ (see Definition 1.5 on page 10). For deductions between sets of sentences, $\Phi_1 \vdash_{\Sigma} \Phi_2$ means $\Phi_1 \vdash_{\Sigma} \varphi$ for all $\varphi \in \Phi_2$, where $\Phi_1, \Phi_2 \subseteq \mathbf{Sen}^{\mathcal{I}}(\Sigma)$.

Structural rules (Figure 5.4).
These rules express the structural properties that any CR according to Definition 1.3 must satisfy, together with some of the properties that a CR for ASL+ must satisfy, from Definition 5.6.

Rules for programs (Figure 5.5).
These rules are used to prove that a program satisfies a specification, or that two programs are equivalent. The figure contains a only a few example rules and is far from complete. Sentences $P : SP$ here abbreviate $P = P : SP$.

The first rule says that any two Σ-programs satisfy the trivial Σ-specification and are indistinguishable at that type; this fulfills the "top equality" requirement from Definition 5.6. The next few rules are structural rules for some

[8]But similar presentations are commonly used for categorical logics, for examples see Lambek and Scott [1986], Pitts [1996]. And a similar presentation is also used for the logic LFPC in Section 6.4.

$$\frac{\Gamma \blacktriangleright s \Rightarrow \Sigma}{s \vdash^{sat}_{\Gamma} s}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} s \qquad \Delta, s \vdash^{sat}_{\Gamma} s'}{\Delta \vdash^{sat}_{\Gamma} s'}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} s \qquad \Gamma \blacktriangleright s' \Rightarrow \Sigma}{\Delta, s' \vdash^{sat}_{\Gamma} s}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma,X:A,\Gamma'} s \qquad \Gamma \blacktriangleright A \Rightarrow \mathsf{P}(\tau) \qquad \Gamma \blacktriangleright N \Rightarrow \tau}{\Delta[N/X] \vdash^{sat}_{\Gamma,\Gamma'} s[N/X]}$$

$$\frac{\Gamma \blacktriangleright SP \Rightarrow \mathsf{P}(\Sigma)}{\vdash^{sat}_{\Gamma} SP \leq SP}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} P = P' : SP}{\Delta \vdash^{sat}_{\Gamma} P' = P : SP}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} P = P' : SP \qquad \Delta \vdash^{sat}_{\Gamma} P' = P'' : SP}{\Delta \vdash^{sat}_{\Gamma} P = P'' : SP}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} SP_1 \leq SP_2 \qquad \Delta \vdash^{sat}_{\Gamma} SP_2 \leq SP_3}{\Delta \vdash^{sat}_{\Gamma} SP_1 \leq SP_3}$$

$$\frac{\Delta \vdash^{sat}_{\Gamma} P : SP \qquad \Delta \vdash^{sat}_{\Gamma} SP \leq SP'}{\Delta \vdash^{sat}_{\Gamma} P : SP'}$$

Figure 5.4: ASL+ proof system — structural rules

$$\frac{\Gamma \blacktriangleright P, P' \Rightarrow \Sigma}{\models^{sat}_{\Gamma} P = P' : \Sigma}$$

$$\frac{\Delta \models^{sat}_{\Gamma} P : SP \qquad \Gamma \blacktriangleright P \Rightarrow \Sigma' \qquad \sigma : \Sigma \to \Sigma'}{\Delta \models^{sat}_{\Gamma} R_{\sigma}[P] : D_{\sigma}[SP]}$$

$$\frac{\Delta \models^{sat}_{\Gamma} P : SP \qquad \Gamma \blacktriangleright P \Rightarrow \Sigma \qquad \Phi' \subseteq \mathbf{Sen}^{\mathcal{I}}(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\Delta \models^{sat}_{\Gamma} E_{\Phi',\sigma}[P] : I_{\Phi'}[T_{\sigma}[SP]]}$$

$$\begin{tikzcd}
\Sigma_1 \arrow[r, dashed, "\sigma_2{*}"] & \Sigma^* \\
\Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 \arrow[u, dashed, "\sigma_1{*}"']
\end{tikzcd}$$

$$\frac{\begin{array}{ll} \Delta \models^{sat}_{\Gamma} P_1 : SP_1 & \Gamma \blacktriangleright P_1 \Rightarrow \Sigma_1 \\ \Delta \models^{sat}_{\Gamma} P_2 : SP_2 & \Gamma \blacktriangleright P_2 \Rightarrow \Sigma_2 \end{array}}{\Delta \models^{sat}_{\Gamma} U_{\sigma_1,\sigma_2}[P_1, P_2] : U_{\Sigma^*}[\,T_{\sigma_2^*}[SP_1],\ T_{\sigma_1^*}[SP_2]\,]}$$

$$\frac{\begin{array}{lll} & \Phi'_1 \vdash^{\mathcal{I}}_{\Sigma'} \Phi'_2 & \sigma : \Sigma \to \Sigma' \\ \Delta \models^{sat}_{\Gamma} E_{\Phi'_1,\sigma}[P] : SP & \Phi'_2 \vdash^{\mathcal{I}}_{\Sigma'} \Phi'_1 & \Phi'_2 \subseteq \mathbf{Sen}^{\mathcal{I}}(\Sigma') \end{array}}{\Delta \models^{sat}_{\Gamma} E_{\Phi'_1,\sigma}[P] = E_{\Phi'_2,\sigma}[P] : SP}$$

$$\frac{\begin{array}{ll} \Delta \models^{sat}_{\Gamma} P_1 : SP & \\ \Delta \models^{sat}_{\Gamma} P_2 : SP & \forall \varphi \in \Phi.\, \Delta \models^{sat}_{\Gamma} P_1 : I_{\{\varphi\}}[\Sigma] \iff \Delta \models^{sat}_{\Gamma} P_2 : I_{\{\varphi\}}[\Sigma] \end{array}}{\Delta \models^{sat}_{\Gamma} P_1 = P_2 : A_{\Phi, id_{\Sigma}}[SP]}$$

Figure 5.5: ASL+ proof system — programs

PBOs, breaking down the structure of a program in tandem with the structure of a specification. To prove that a program satisfies a specification with a different structure, we may first need to massage the structure of the specification, using the rules in Figure 5.6, and then use subsumption.

Two simple rules proving non-trivial program equivalence are shown. The first uses the proof system of $\mathcal{I}$ to show that two free extensions $E_{\Phi'_1,\sigma}[P]$ and $E_{\Phi'_2,\sigma}[P]$ are equivalent just in case Φ'_1 and Φ'_2 are equivalent sets of equations in the logic. By this rule, a program expressed using one algorithm can be proved equivalent to one expressed using another slightly different one, for example.

The other rule for program equivalence, at the bottom of the table, is a putative rule intended to prove the equivalence of two programs in a specification built with **abstract**. This attempts to capture a simple case of the definition of $\equiv^{\Phi}_{\sigma}$, with $\sigma = id$. We can prove $P_1 \vdash_{\Sigma} \varphi$ by proving $P_1 : I_{\{\varphi\}}[\Sigma]$ in the system, and similarly for P_2. However, this rule breaks the definition of $\models^{sat}$ because the relation occurs negatively in the premise. This could be solved by using a direct definition of $P \vdash_{\Sigma} \varphi$, written using another set of rules. Still this rule doesn't represent a feasible proof *method*, since Φ is typically an infinite set. We need to investigate proof techniques here.

Rules for specifications (Figure 5.6).
These rules are example rules for proving refinement of specifications. Some of them are stolen from a more complete set studied by Farrés-Casals [1992]. In this figure, a sentence $SP = SP'$ abbreviates the two sentences $SP \leq SP'$ and $SP' \leq SP$ — used in a conclusion it abbreviates two rules.

The first rule in Figure 5.6 establishes the "top refinement" property, that any Σ-specification is a refinement of the trivial specification Σ. The next rule is the only rule which refers to the underlying proof system. The following axioms can be used for proving refinements or massaging the structure of specifications. The last rule in the table is an example of a simplification rule, which proves refinement between specifications with the same outermost constructor. Simplification by stripping the outermost constructors is sound for any of the familiar ASL constructors, because they are monotonic on the model class inclusion ordering.

Many more rules from Farrés-Casals [1992] could be included here, including proofs of refinements between specifications with differing structure, and more simplification rules. And rules are needed to handle the SBOs missed here, in particular **abstract** and **minimal**.

$$\frac{\Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma)}{\models^{sat}_{\Gamma}\ SP \le \Sigma}$$

$$\frac{\Phi_1 \ \vdash^{1}_{\Sigma}\ \Phi_2 \qquad \Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma) \qquad \Phi_1, \Phi_2 \subseteq \mathbf{Sen}^{1}(\Sigma)}{\models^{sat}_{\Gamma}\ I_{\Phi_1}[SP] \le I_{\Phi_2}[\Sigma]}$$

$$\frac{\Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma) \qquad \Phi_1, \Phi_2 \subseteq \mathbf{Sen}(\Sigma)}{\models^{sat}_{\Gamma}\ I_{\Phi_1}[I_{\Phi_2}[SP]] = I_{\Phi_1 \cup \Phi_2}[SP]}$$

$$\frac{\Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma) \qquad \Phi \subseteq \mathbf{Sen}(\Sigma) \qquad \sigma : \Sigma \to \Sigma'}{\models^{sat}_{\Gamma}\ I_{\Phi}[D_{\sigma}[SP]] = D_{\sigma}[I_{\sigma(\Phi)}[SP]]}$$

$$\frac{\Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma) \qquad \Phi \subseteq \mathbf{Sen}(\Sigma) \qquad \sigma : \Sigma \to \Sigma'}{\models^{sat}_{\Gamma}\ I_{\sigma(\Phi)}[T_{\sigma}[SP]] \le T_{\sigma}[I_{\Phi}[SP]]}$$

$$\begin{tikzcd}
\Sigma_1 \arrow[r, dashed, "\sigma_2{}^*"] & \Sigma^* \\
\Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 \arrow[u, dashed, "\sigma_1{}^*"']
\end{tikzcd}$$

$$\frac{\Gamma \blacktriangleright SP \implies \mathsf{P}(\Sigma)}{\models^{sat}_{\Gamma}\ T_{\sigma_1}[D_{\sigma_2}[SP]] = D_{\sigma_1^*}[T_{\sigma_2^*}[SP]]}$$

$$\frac{}{\models^{sat}_{\Gamma}\ U_{\Sigma}[SP_1, SP_2] = U_{\Sigma}[SP_2, SP_1]}$$

$$\frac{}{\models^{sat}_{\Gamma}\ U_{\Sigma}[SP_1, U_{\Sigma}[SP_2, SP_3]] = U_{\Sigma}[U_{\Sigma}[SP_1, SP_2], SP_3]}$$

$$\frac{\Delta \models^{sat}_{\Gamma}\ SP_1 \le SP_2}{\Delta \models^{sat}_{\Gamma}\ D_{\sigma}[SP_1] \le D_{\sigma}[SP_2]}$$

Figure 5.6: ASL+ proof system — specifications

5.6.1 Properties of the ASL+ proof system

By the construction of the system, all sentences are well-typed.

Proposition 5.13. *If $\Delta \models^{sat}_{\Gamma} s$, where $\Delta = \{ s_1, \dots, s_n \}$, then there are signatures $\Sigma_1, \dots, \Sigma_n, \Sigma$ such that $\Gamma \blacktriangleright s_i \Rightarrow \Sigma_i$ and $\Gamma \blacktriangleright s \Rightarrow \Sigma$.*

Proof Induction on the derivation of $\Delta \models^{sat}_{\Gamma} s$. □

The two vital properties are that $\models^{sat}$ forms a consequence relation for $\lambda_{\mathsf{ASL+}}$, and that it is sound for the semantics. I only give proof sketches.

Proposition 5.14. *$\models^{sat}$ is a $\lambda_{\mathsf{ASL+}}$ consequence relation according to Definition 5.6.*

Proof Most requirements follow directly from the conventions of the presentation and the structural rules of Figure 5.4. Remaining requirements are admissible in the presentation. □

Proposition 5.15. *$\models^{sat}$ is a sound consequence relation for the $\lambda_{\mathsf{ASL+}}$ semantics, according to Definition 5.11.*

Proof By induction on derivations of $\models^{sat}$, using the semantics of the ASL+ operators defined in Sections 5.4.2 and 5.4.3. □

5.7 Improvements Over the Original

This section compares the formulation of ASL+ given in this chapter against the original proposal by Sannella et al. The purpose is to highlight the improvements I have made and to justify the definitions a bit more. (Readers not interested in this purpose may happily proceed to Section 5.8 on page 170.)

ASL+ was first described in [Sannella et al., 1992, Sannella and Tarlecki, 1991], and earlier versions of those publications. The original syntax was outlined in Section 1.4 on page 19. The next few subsections describe some problems with the original syntax and proof system, and what I have done to improve matters.

5.7.1 Unspecified syntax for algebras

The original presentation of ASL+ includes *unspecified* syntax for core-level programs denoting algebras in the base institution. Syntactic algebras can be type-checked, so there should be a type-checking judgement:

$$\Gamma \blacktriangleright P \Rightarrow \Sigma$$

where Γ is a context declaring the free variables which may appear in P, along with their types. If this judgement holds, then P denotes a Σ-algebra. Sannella et al. required that this judgement be *substitutive*, so we can infer:

$$\frac{\Gamma, X : \tau, \Gamma' \blacktriangleright P \Rightarrow \Sigma \qquad \Gamma \blacktriangleright N \Rightarrow \tau}{\Gamma, \Gamma' \blacktriangleright P[N/X] \Rightarrow \Sigma}$$

where N is any term of the module calculus, and τ is the typing requirement it must meet.

Further assumptions of the algebra typing judgement would be needed, actually, to prove the properties which were shown in Section 5.5.3. Moreover, we need an assumption relating the denotation of a substituted algebra expression to its denotation in an extended environment (like Lemma 3.21 on page 72). These assumptions are needed for the algebra typing judgement and semantics, to prove the corresponding properties for the whole calculus.

Here is the difficulty. The assumptions about substitution imply that the language of algebras *includes* the language of ASL+, so that both the syntax and semantics of the two levels are already intermingled. The term N above can be any term of ASL+, so we must know how to substitute N into an algebra P for the term $P[N/X]$ to be meaningful. But this destroys the thought that we may begin from an arbitrary core-level language and add the ASL+ λ-calculus on top. For terms like $P[N/X]$ to be meaningful, the core-level syntax must be defined simultaneously with the ASL+ syntax. This means that ASL+, as originally proposed, could not be an institution-independent framework, unless we have a rather more concrete form of institution to hand.

To avoid this difficulty, I began with a more precise language. Instead of giving the core-language for programs by some unspecified syntax with particular properties, it is given by a set of combinators (PBOs) which denote mappings from tuples of algebras to algebras. Such functions (or their extensions to mappings on classes of algebras) are known as *constructors* in the literature [Sannella and Tarlecki, 1988b]. Similarly, rather than fix the particular specification building operators of ASL, I treated core-level specifications as an abstract set of SBOs, which are combinators that denote operations on classes of algebras. With this more precise language, it is easy to include the syntax of fully-applied combinators in the ASL+ λ-calculus, solving the problem of what substitution means.

5.7.2 Semantic inference rules

Another difficulty with the original version of ASL+ is with the rules for so-called *semantic inference*. These were given as part of the formal system for proving satisfaction:

$$\frac{[\![P]\!]_\rho \in [\![SP]\!]_\rho \text{ for all } \rho \text{ consistent with } \Gamma}{\Gamma \rhd P : SP}$$

$$\frac{[\![SP]\!]_\rho \subseteq [\![SP']\!]_\rho \text{ for all } \rho \text{ consistent with } \Gamma}{\Gamma \rhd SP : \mathit{Spec}(SP')}$$

These rules were intended to apply only to simple programs and specifications — those built with the core programming language, or with the ASL specification building operators. The idea is that an external proof system would be employed to approximate the use of the semantics in these rules. Unfortunately, this requirement is too informal.

Once again, "simple" programs and specifications may be built using constructs of ASL+ within, which makes them somewhat less simple. For example, we may write:

$$\textbf{impose } \Phi \textbf{ on } F(M)$$

where $F(M)$ is the application of a parameterised specification. To prove that a program satisfies this specification, we should first prove that it satisfies Φ, and then that it satisfies $F(M)$. To prove the second satisfaction, we ought to use higher-order rules for application terms which are proper to ASL+, rather than only the "semantic inference" rule shown above.

The study of proofs in the original formulation is doomed because of this. Using the rules above and bypassing the rules that are proper to ASL+, we could prove *any* valid satisfaction $P : SP$ by proving

$$P : \textbf{impose } \varnothing \textbf{ on } SP$$

and *any* valid refinement $SP \le SP'$ by proving

$$\textbf{impose } \varnothing \textbf{ on } SP : \mathit{Spec}(\textbf{impose } \varnothing \textbf{ on } SP').$$

Clearly the rules of semantic inference had to be modified.

In this case, I solved the problem by bringing the external proof system into the picture, making the formal system for satisfaction more precise. Instead of referring to the semantics, a consequence relation $\vdash^{sat}$ must be provided. The consequence relation captures derivability in the external proof system.

For this treatment to work, the consequence relation must be *schematic*, to allow substitution for complex ASL+ terms inside simple programs and specifications. For example, to prove satisfaction of the specification above, the consequence relation would contain the tuples

$$P : X \vdash^{sat} P : \textbf{impose } \Phi \textbf{ on } X$$

for all simple programs P that satisfy the formulae Φ. Then we use the cut-like rule (PBO) for joining this assertion with the other rules of $\lambda_{\mathsf{ASL+}}$, to allow the derivation:

$$\frac{P : X \vdash^{sat} P : \textbf{impose } \Phi \textbf{ on } X \qquad P : F(M)}{P : \textbf{impose } \Phi \textbf{ on } F(M)}$$

To apply this rule I used a substitution $X \mapsto F(M)$ to instantiate a tuple in the consequence relation $\vdash^{sat}$. The antecedent is then proved in the second premise, using the rules of $\lambda_{\mathsf{ASL+}}$ once more.

Of course, the program P itself might be built using ASL+ constructs too, so the consequence relation must contain schematic consequences which allow programs as well as specifications to be broken down. In other words, it captures the derivation relation of a *structural proof system*, rather than a proof system which only proves entailments between normal forms. Moreover, as well as satisfaction $P : SP$, the consequence relation must prove refinement $SP \leq SP'$ and equality of two programs satisfying a specification $P = P' : SP$. Since equality should be reflexive, we used $P : SP$ to abbreviate $P = P : SP$.

5.7.3 Other improvements

For the historical record, here are some more notes about the formalism presented in the last section of Sannella et al. [1992] and differences with the work here. Some of these comments are technical and esoteric.

Semantics. The definition of the semantics here has been structured using rough types, as explained in Sections 4.5 and 5.2.2, and as suggested in Sannella et al. [1992]. In loc.cit., the semantics was given using a partial definition with slightly informal and "non-local" side conditions; in the clause for $[\![\lambda X{:}SP.\, Obj]\!]_\rho$, the side condition is "provided for each $v \in [\![SP]\!]_\rho$, $[\![Obj]\!]_{\rho[v/X]}$ is defined." This is a stricter version of the partial function model outlined in Example 4.30; there, the λ-term is interpreted as a function which could return an error element when applied to some $v \in [\![SP]\!]_\rho$, and is not undefined whenever the body is undefined for some v. On terms which are typable in $\lambda_{\mathsf{ASL+}}$, there is no difference between these partial definitions and the full model which *ignores* the domain specification SP in the interpretation for λ, and just uses rough types.

Reduction rules. The presentation in loc.cit. uses a "split" rule of conversion for types based on untyped reduction, as mentioned in Section 4.4.2. The combination of untyped reduction rules and singleton types *without* type tags, has some strange effects. In particular, one can derive:

$$\dfrac{\dfrac{\dfrac{M' : A}{M' : \{M'\}} \qquad \{M'\} \twoheadrightarrow_\beta \{M\}}{M' : \{M\}} \qquad \dfrac{M : SP}{\{M\} : \mathsf{Spec}(SP)}}{M' : SP}$$

This shows the derivability of a rule allowing well-formed β-expansion of terms satisfying a specification:

$$\dfrac{M : SP \qquad M' \twoheadrightarrow_\beta M \qquad M' : A}{M' : SP}$$

Embarrassingly, Sannella et al. stated that they had specifically excluded this rule from the system! This goes to show that the interaction of "non-informative" type constructors can be subtle.

The reason for not wanting this rule is methodological: ideally, we should prove $(\lambda X{:}A.\,M)N : SP$ in a structural way, not simply by β-reduction and proving $M[N/X] : SP$. But the methodological aspects of the system should not be confused with the logical aspects; practical implementations can enforce a particular proof search mechanism if required.

Nevertheless, the derivability of the rule above is still perturbing, and motivates using singletons with type tags and a semantic-oriented presentation instead of untyped reduction. See the discussion in Section 4.4.2.

Subject reduction. Sannella et al. [1992, Lemma 7.3] claimed subject reduction for their system. Unfortunately the sketched proof in Sannella and Tarlecki [1991] was incomplete and is non-trivial to fix, see Section 4.4.1. This problem is still open.

Other changes. The original semantics used sets rather than PERs; the original proof system used the Π rule of equal domains (see Section 4.9.2).

5.8 Two Problems

As I mentioned in Section 5.1, there are a couple of problems with this formulation of ASL+, which prevent it being a directly useful module language.

The first is the **context problem**. This problem is that the PBO and SBO mechanism does not provide a framework for understanding a current context of declarations, or a current environment of bindings. A good way of dealing with contexts is essential to the realistic use of the language, as soon as we go beyond "closed" programs or specifications, or beyond simple examples of parameterisation with only one parameter. We consider this problem in Section 5.8.1.

The second problem is that rough types are not good enough to typecheck programs and specifications like those shown in Chapter 2. The related fact is that those programs do not have a direct semantics in an institution. This is the **sharing problem** — to explain the propagation of type equalities properly. This is investigated in the rest of this chapter, beginning in Section 5.8.2.

5.8.1 *The context problem*

The treatment of PBOs and SBOs as combinators is not very satisfactory for constructing modular systems.

To write the parameterised program `List` and its specification `LISTFUN` shown in Section 2.1.3 on page 28, we begin with the signatures:

$$\Sigma_{ELT} =_{\mathrm{def}} \textbf{sig} \ \ \textbf{type}\ \mathtt{elt}\ \ \textbf{end} \qquad \Sigma_{LIST} =_{\mathrm{def}} \begin{array}{l}\textbf{sig}\\ \quad \textbf{type}\ \mathtt{elt}\\ \quad \textbf{type}\ \mathtt{list}\\ \quad \textbf{val}\ \mathtt{nil} : \mathtt{list}\\ \quad \textbf{val}\ \mathtt{cons} : \mathtt{elt} \times \mathtt{list} \to \mathtt{list}\\ \quad \vdots\\ \textbf{end}\end{array}$$

In the abstract formulation of ASL+ given in the last section, Σ_{ELT} and Σ_{LIST} could be represented by nullary SBOs (corresponding to the syntax above), and `List` might be defined as

$$\mathtt{List} =_{\mathrm{def}} \lambda X{:}\Sigma_{ELT}.\, \textbf{extend}\ X \text{ to } \Phi_{list}\ \textbf{via}\ \iota_{\Sigma_{ELT},\Sigma_{LIST}}$$

where Φ_{list} are the axioms specifying lists, shown in the specification `LISTFUN` on page 28. The specifications `LISTFUN` and `LIST` would be written similarly. The three terms all have the general form:

$$\overset{\lambda}{\underset{\Pi}{}} X \ : \ \mathit{PARAMETERTYPE} \ . \ \mathit{EXTENDOPERATOR}(\,X,\ \mathit{extrabits}\,)$$

(remember that to explain the dot notation, `LISTFUN` has to be translated to use singleton types and **enrich**, as explained in Section 2.1.5).

This idiom is prevalent in the examples in Chapter 2 and the examples of ASL+ elsewhere. In each case, the parameter is used in a standard way to build the result. When the parameter is a *specification*, as in `LIST`, then there is more choice: we might use an operator for closing up the parameter under some equivalence, for example, before enriching it. But when the parameter is a *program*, as in `LISTFUN` and `List`, we intend a *constructive* interpretation, where a new program is built (or specified) from the parameter without altering it; in other words, the body defines a *persistent* constructor [Sannella and Tarlecki, 1988b].

This idea lies behind the dot notation used in `LISTFUN`. There is an implicitly-available algebra which represents the current context. When programming, it is more convenient to use pieces of parameters with the dot notation than to each time explicitly extend, and perhaps rename, the parameter. This is similar, although not quite as extreme, to the difference between programming in λ-calculus and combinatory logic; in the second case one can only use closed terms. Here, we have open terms at the module-level terms, but closed terms for core-level signatures and algebras. The `List` example above is readable, but once many parameters are added, things get worse, piling extension on extension, without any name-space management unless renamings are supplied explicitly by the user.

Because the most important parameters in the examples were all algebras, it seems worthwhile trying to do something about this. We can treat algebra parameters specially, and introduce a form of context which allows algebras and signatures to be open. Then this will also explain the implicit use of pervasive datatypes, like the algebra of booleans.

To do this, the idea of combinators for building programs is not so good, because we want dot notation to refer to *parts* of algebras. It seems better to return to something like the original assumption described in Section 5.7.1: having some unspecified syntax which respects substitution operations, so there is a closer link to the concrete syntax used for simple programs and specifications. But restrictions should be made so that the free variables in the unspecified syntax only range over "core-level" entities. We want a notion of an algebra or signature *fragment*, which is meaningful in a context. The context for simple programs and specifications should only mention core-level entities, so there should be a way of extracting a core-level context from an ASL+ context. Moreover, we would like to integrate a good way of handling the dot notation used in Chapter 2, to project from the components of a context.

These are some motivations behind the approach taken in Chapter 6 and Chapter 7, using the institution $\mathcal{FPC}$. The ideas outlined above will become clearer in these later chapters; a context for defining $\mathcal{FPC}$ algebras and signature expressions is simply an $\mathcal{FPC}$ signature, and it is built by renaming components with a dot-prefixing operation.

5.8.2 The sharing problem

An appealing view of modular programming is to treat the putting-together of programming modules as typed functional programming [Burstall, 1984]. This is based on an analogy between interfaces and types, and program modules and values. So a module whose interface contains other modules is understood as a function from modules to modules.

Unfortunately, the "rough types" used for type-checking in ASL+ are not good enough to understand the composition of parameterised programs and specifications in this way. For example, the functor `List` shown above is given the rough type:

$$\langle\rangle \blacktriangleright \texttt{List} \implies \Sigma_{ELT} \to \Sigma_{LIST}$$

which reflects that it maps Σ_{ELT}-algebras to Σ_{LIST}-algebras.

There are two related difficulties. The crucial one is that the rough type $\Sigma_{ELT} \to \Sigma_{LIST}$ makes no connection between result of applying `List` and the argument algebra, although from the definition of `List` we know that the result is always a Σ_{LIST}-expansion of the argument. The semantics of the rough type reflects this lack of connection: the function space $\Sigma_{ELT} \to \Sigma_{LIST}$ includes constant functions which ignore their argument entirely.

The other difficulty is that the type $\Sigma_{ELT} \to \Sigma_{LIST}$ only allows `List` to be applied to Σ_{ELT}-algebras: in practice we may want to use an algebra such as `Nat` which has the richer signature:

```
ΣNAT =def sig
            type elt
            val zero : elt
            val succ : elt → elt
          end
```

To apply `List` to `Nat` we first need to cut it down to size, writing something like:

`ListNat` $=_{\text{def}}$ `List` (**reduct of** `Nat` **by** $\iota_{\Sigma_{ELT},\Sigma_{NAT}}$)

The rough type for this instance of **reduct** is:

$$\langle\rangle \blacktriangleright \textbf{reduct of } - \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{NAT}} \implies \Sigma_{NAT} \to \Sigma_{ELT}$$

which, like the type of `List`, makes no connection between argument and result.

Because there is no connection, rough typing information alone is not enough to type-check program-level and specification-level terms when modules are combined. Suppose we rename ListNat by prefixing Σ_{LIST} with "ListNat." and rename Nat by prefixing Σ_{NAT} with "Nat.". This renaming might happen automatically with a notion of a current context of declarations; when the two modules are combined the signature of the context becomes the union of these two signatures, $\Sigma_{ctx} = \mathtt{Nat.}\Sigma_{NAT} \cup \mathtt{ListNat.}\Sigma_{LIST}$. In this signature, we would hope to write a formula such as this:

$$\mathtt{ListNat.hd(ListNat.cons(Nat.zero, ListNat.nil))} = \mathtt{Nat.zero}$$

To know that this formula is well-typed, we must know the type equation ListNat.elt = Nat.elt. But nothing in rough type of List entitles us to suppose that these two types are equal, and they are distinct sorts in Σ_{ctx}.

The ASL+ *specification* (or "full type") for List is much better; as well as imposing axioms for the list functions in the result, a dependent type and the singleton construct specify that the result should be an expansion of the input:

$$\mathtt{List} : \Pi X{:}\Sigma_{ELT}.\,\textbf{impose } \Phi_{LIST} \textbf{ on translate } \{X\} \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{LIST}}$$

The idea of using dependent types to express modular structure appeared in Pebble [Lampson and Burstall, 1988] and was explained by MacQueen [1986].

The Π-type for List correctly cuts down the range of permissible implementations to those which make use of their parameter. With suitable rules for reasoning about the renaming SBOs, we might now *prove* the equations:

$$\mathtt{ListNat.elt} = (\textbf{reduct of } \mathtt{Nat} \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{NAT}}).\mathtt{elt}$$

and

$$(\textbf{reduct of } \mathtt{Nat} \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{NAT}}).\mathtt{elt} = \mathtt{Nat.elt}$$

which justify writing the formula above. To deduce the second equation, we need to know that **reduct of** – **by** $\iota_{\Sigma_{ELT},\Sigma_{NAT}}$ satisfies the specification

$$\textbf{reduct of } - \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{NAT}} : \Pi X{:}\Sigma_{NAT}.\,\textbf{derive from } \{X\} \textbf{ by } \iota_{\Sigma_{ELT},\Sigma_{NAT}}$$

and then we use the putative rules about renamings again.

But this is not satisfactory — we want to deal fully with *type-checking* in the rough typing system, automatically if possible, and not have to deal with it in the *satisfaction* system when human intervention is perhaps needed (depending on the logic).

Even worse, because rough typing fails, the obvious, direct institutional semantics for the constructs above does not work. If Σ_{ctx} is the signature mentioned above, in a typical institution, the set of sentences $\mathbf{Sen}(\Sigma_{ctx})$ only consists of "well-typed" sentences over Σ_{ctx}, and so does not contain the formula above, because `ListNat.elt` and `Nat.elt` are distinct sort names.

Somehow, we must either rename names which share to a single name (similar to a pushout semantics, taking amalgamated unions of signatures), or we must add *sharing equations* between the type names. Pushouts serve at the semantic level, but names in semantic signatures need to be related to the program-level identifiers which the user writes during modular construction of programs. Identification of names is also less flexible than sharing equations with type expressions, as mentioned in Section 2.1.5.

One way or another, the sharing information must be known to type-check terms, hence programs, formulae and specifications. But the rough types used so far cannot provide this information.

I draw two conclusions from these observations:

1. *Simple rough types are not sufficient to capture the "type-checking" part of putting together modules in ASL+.*

2. *Ordinary algebraic signatures are not sufficient to capture sharing between sort names needed for type-checking with program-level identifiers.*

The next two sections are taken up by more consideration and justification of these conclusions. In Section 5.9 we see where the problem of sharing arises when constructing modular programs, and look at different ways of expressing sharing. In Section 5.10 we consider ways of incorporating sharing into an institutional semantics, and argue that the best approach is to modify signatures to incorporate sharing information. This will be done in the institution $\mathcal{FPC}$ defined in Chapter 6.

The first conclusion will be confronted in Chapter 7 when a more sophisticated rough typing system is defined for the institution $\mathcal{FPC}$, incorporating dependent types.

5.9 Sharing in Modular Programs

The need for ways of expressing sharing in modular programs has been understood for some time, probably long before MacQueen's seminal exposition on the subject written during the development of SML [MacQueen, 1986]. However, it seems worth re-investigating the basics here, to understand the alternatives and carefully motivate extensions to ASL+.

Aliasing

Aliasing is a well-known difficulty when trying to formally understand programming languages. Flexible methods of name binding and parameter passing mean that the same program entity (type, value, module, or whatever) can be referred to via several different identifiers, even within the same scope. Conversely, the same identifier may be used in different scopes to refer to different entities. The problem is to keep track of the proliferation and variation of identifiers. This is necessary for giving a formal semantics, a program logic, or perhaps a type-system to the programming language.

Aliasing can be handled by converting any identifier to a canonical one by a so-called "unique origin" rule, or by associating each identifier with some other reference value. For example, the 1990 formal semantics of Standard ML [Milner et al., 1990] associates unique semantic-level stamps (called *names*) to types and modules, when defining compile-time type-checking. Type identifiers may be bound to different names according to the scope.

Although aliasing is rife in programming languages, in logics and type theories it is sometimes less apparent. This is partly because of the complexity of programming language features, but also partly because logics and type theories are studied in the abstract: for example, *definitions* are typically dealt with in the informal meta-language rather than in the object language itself. Yet when given a formal treatment, definitions turn out to be less tractable than might be hoped [Griffin, 1988, Harper and Pollack, 1991, Severi and Poll, 1994]. Researchers applying type theory are currently adding such "high-level" concepts to the formal part of the theory, both to understand their implementation, and to bring the mechanical language gradually closer to the looser language of the mathematician. And as the implementation of mathematical theories becomes more advanced, new issues of modularisation and re-use need to be dealt with, and this brings in further problems of name-space management and aliasing.

Modular programming

An important case of aliasing occurs when building modular programs [MacQueen, 1986]. In this setting, when two identifiers for modules (or other entities) stand for the same thing, we say that they *share*. This is because in the compiled program, the two names really share the same implementation. (Whether the compiler reuses the same *code* is a slightly different issue: for optimization it might be desirable to duplicate code by in-lining, for example.) As well as type-checking, we need to know about sharing for constructing proofs about specifications and programs.

Sharing can be split into two kinds. First, for the combination of two program modules to make sense, some subparts may have to share the same implementation. Two modules which communicate using sets of integers, for example, are only compatible if their underlying implementations of integer sets are the same. Second, when a generic or parameterised module is instantiated, part of the result may share with part of the argument. For example, when a module which yields binary trees of sets of integers is given an input module describing sets of integers, we might like to construct trees using previously constructed sets of integers (to give a more realistic version of the `ListNat` example from Section 5.8.2). Again, the underlying implementations must be the same.

In the next few subsections, I will elaborate on the two kinds of sharing mentioned above: *sharing in the parameter*, and *parameter-result sharing*. I will also explain the need to propagate "extra" sharing from the argument — *argument-result sharing* — and how *sharing by parameterisation* can be used with higher-order modules.

5.9.1 Sharing in the parameter

Plugging modules together by parameterisation, we may need to know that there is some sharing between the modules to be used as arguments; for example, that two of them will have the same implementation of integer sets.

Sharing in the parameter amounts to requiring sharing amongst some parts of the context, where the context is a collection of assumptions about the interfaces of modules. Typically we need to *assume* this kind of sharing for the formal parameters of a parameterised module.
In SML, sharing in the parameter appears in functor headings like this:

```
functor F (structure S1 : sig type intset ... end
           structure S2 : sig type intset ... end
           sharing type S1.intset = S2.intset)
  = ...
```

MacQueen [1986] calls this the "diamond import" situation, because of the picture:

$$\begin{tikzcd}
& \mathtt{intset} \arrow[dl] \arrow[dr] & \\
\mathtt{S1} \arrow[dr] & & \mathtt{S2} \arrow[dl] \\
& \mathtt{F} &
\end{tikzcd}$$

This means that F should only be applicable to structures S1 and S2 which have the same implementation of intset.

5.9.2 Parameter-result sharing

When a generic or parameterised module is instantiated, we may need to know that part of the result shares with the argument that was used as an instantiation.

Parameter-result sharing is the kind of sharing that occurs between some parts of a module and its context. Typically this occurs when a program entity is propagated from the import interface of a parameterised module to the export interface, so the body of the parameterised module *constructs* this kind of sharing from entities in the formal parameter.
In SML, this type of sharing appears in functor headings like this:

```
functor G (structure S : sig type intset ... end)
        : sig type t ... sharing type t = S.intset end
  = ...
```

This heading tells us that the functor G must propagate the type intset from the argument to the type t in the result.

Perhaps only pedantic SML programmers write parameter-result sharing, because the SML type-checker automatically infers it from the body of the functor, which is always written under the functor heading in SML. In other languages where the functor body is not available, one needs to carefully specify parameter-result sharing. One case is Extended ML [Kahrs et al., 1994], where functor bodies can be unimplemented; other cases occur when there is a higher-order parameterisation or mechanisms for separate compilation, when again functor bodies can be unknown. In ASL+, parameter-result sharing is needed because of higher-order parameterisation.

5.9.3 Argument-result sharing

When an argument is supplied to a parameterised module, it may witness "extra sharing" over and above the requirements expressed in the formal parameter. Rather than forgetting this extra sharing, the type-checker typically propagates it to the result of applying the parameterised module.
For example, in SML if we have a functor

```
functor F (structure S:sig type t end) : sig type t=S.t end
 = ...
```

then we can apply F to a structure with a richer signature, for example,

```
structure T = F(struct type t=int end)
```

The inferred signature of the argument structure here expresses the fact that t shares with int, which is a name from the context. Although this sharing equation is not requested in the parameter signature of F, we might reasonably expect the equality to be propagated, so that T.t = int. Indeed, this behaviour turns out to be invaluable in practice.

Perhaps it seems strange to imagine that this "extra sharing" would not be propagated: reasoning equationally, if F(S).t = S.t and S.t = int, it follows that F(S).t = int. However, this is a design choice, because of the different ways of treating type equality in programming languages. In type theories we expect types to be compared with a *structural equality*, but in programming languages it is more usual to use *name equality*. Name equality is efficient to implement, and moreover, allows type abstraction by preventing confusion of types which "accidentally" have the same structure. With name equality, we have a choice over whether to propagate extra sharing or not.

The standard *forgetful* explanation of how to apply parameterised specifications in ASL or ASL+ corresponds roughly to generating new type names when a parameterised module is applied.[9] This standard explanation is that the **derive** operator or reduct function must be used to forget any extra information present in the argument signature, as shown for the example ListNat on page 173. If sharing information about type identities is part of the signature, as will be proposed in Section 5.10, it is forgotten by the **derive** operation and cannot be propagated to the result.

The crux here is that parameterised programs and specifications in ASL+ are given *fixed* input and output signatures. For example, a parameterised

[9]There is still a question over whether *each* application of a module produces new names, or whether every application of a module to the same argument produces the same names: this is the distinction between a *generative* and *applicative* semantics mentioned in Section 7.3.5.

program denotes a single function,

$$f : \mathbf{Mod}(\Sigma_{par}) \to \mathbf{Mod}(\Sigma_{res}).$$

But in practical module systems, a parameterised program denotes a *family* of functions indexed by the signature of the actual parameter,

$$f_{\Sigma_{arg}} : \mathbf{Mod}(\Sigma_{arg}) \to \mathbf{Mod}(\Sigma_{res} \oplus \Sigma_{arg}),$$

where the result signature $\Sigma_{res} \oplus \Sigma_{arg}$ depends in some way on the signature of the actual argument. This dependence of the output signature upon the input signature is reflected in most explanations of parameterisation in specification languages, too, in particular including the pushout approach widely used since CLEAR [Burstall and Goguen, 1980]. The signature of the output need not contain extra names[10] but it should contain the sharing information from Σ_{arg}, to allow argument-result sharing.

If sharing information is added to the institutional notion of signature, then the rough typing approach in ASL+ cannot be good enough. If sharing is explained outside the institution signatures, then we would need extra annotations to explain the propagation behaviour of parameterised programs. A whole new type system would be needed to type-check modular programs.

5.9.4 Sharing by parameterisation

At least since Pebble [Lampson and Burstall, 1988], researchers have known that sharing in the parameter is not needed when higher-order parameterisation is available. This is because, in the absence of cyclic dependencies, we can re-express examples like the one above by "lifting" out the common part of the parameters and putting it into a new parameter.

Sharing by parameterisation was used heavily in the example in Chapter 2. To re-cap, here is a variation of the first example above (in a putative extension of SML):

```
functor F'
  (structure IntSet:INTSET
   functor FS1(X:INTSET) : sig
                              ...
                              sharing type intset = X.intset
                           end
   functor FS2(X:INTSET) : sig
                              ...
```

[10]Certainly not extra user-level identifiers; additional names from the input could be treated as hidden in the output. See Section 7.3.2.

```
                                     sharing type intset = X.intset
                                 end)
    = struct
        structure S1=FS1(IntSet)
              and S2=FS2(IntSet)
      ...
      end
```

The functor body provides the desired sharing, by construction.

The change from `F` to `F'` adds an extra parameter to the functor and moves the declaration of `S1` and `S2` into the functor body. This makes the modular structure more complicated, but it has important advantages. First, the application of `F'` cannot fail because of accidentally basing `S1` and `S2` on different SET-structures. Second, more freedom is allowed in the order of program development; we may implement `IntSet` and `FS1`, `FS2` in any order.

The crucial point is this: to know `S1.intset=S2.intset` in the body of the functor, *we must know the parameter-result sharing* specified in the functor heading for `FS1` and `FS2`. So using sharing-by-parameterisation, we can reduce sharing in the parameter to parameter-result sharing, but the latter is still needed.

5.9.5 How to provide sharing

The preceding subsections surveyed the different ways that sharing can be constructed or requested in modular programs. The conclusion is that, when higher-order parameterisation is available, there is one kind of sharing which must be explained: sharing between parameter and result. To be a useful way of deriving type identities, the possibility for allowing extra sharing from the argument is also important.

The type system I shall define in Chapter 7 focuses on providing parameter-result sharing, including propagation of extra sharing from the argument.

5.10 Sharing in Institutions

Suppose we wish to extend an institution-based semantics of a specification and programming language to one which provides parameterised specifications and programs. We want a generic construction which works on any particular institution and any simple specification or programming language over it.

Following the model-theoretic approach outlined in Chapter 1, programs are syntactic expressions which denote models from an institution $\mathcal{I}$, and specifications denote classes of models. More fanciful views than this are certainly possible, to bring the syntax of programs and specifications further into the picture. (For example, we can consider a programming language to be an institution in which the satisfaction relation is a function from sentences to models.) But I shan't consider syntax further just yet.

The problem is to account for sharing. While a real language (particularly a programming language) may already include an understanding of sharing, the usual institutional semantics of ASL in an institution such as $\mathcal{EQ}$ or $\mathcal{FOL}$ does not.

What is meant by this? Simply that there is no way to specify sharing in algebraic signatures. For example, in the signature

$$\begin{array}{l} \Sigma =_{\text{def}} \textbf{sig} \\ \qquad \textbf{sorts}\ s, t \\ \qquad \textbf{opns}\ c : s,\ d : t \\ \quad \textbf{end} \end{array}$$

the equation "c=d" is ill-typed and so not in $\mathbf{Sen}(\Sigma)$. But examples in Section 5.9 showed that, either by assumption or by the way the program is constructed, the two sorts s and t might always denote the same set, so the equation $c = d$ *is* type-correct.

There are two basic ways to solve this, corresponding to whether or not we deal with sharing information.

Ignore sharing We could extend the satisfaction relation $\models$ to be three valued, so that $A \models \varphi \in \{\textbf{true}, \textbf{false}, \textbf{wrong}\}$. Then $\mathbf{Sen}(\Sigma)$ is extended to contain all formulae which could possibly have a denotation. So now $c = d \in \mathbf{Sen}(\Sigma)$, but if $A_c \neq A_d$, then $(A \models c = d) = \textbf{wrong}$. This is a bit like *dynamic type-checking* in programming languages.

Handle sharing If we deal with sharing information somehow along with signatures, then we can maintain the idea of *static type-checking.* Then $\mathbf{Sen}(\Sigma)$ consists only of formulae which have a denotation in the semantics, as usual.

The idea of a three-valued or partial satisfaction is interesting, and multiple-valued satisfaction relations have already been studied in the institutions literature. The satisfaction condition makes sense because formulae should not change definedness under a renaming. However, some standard constructions on institutions with two-valued satisfaction relations, such as the addition of negation, need modification.

More importantly, it seems wrong to ignore sharing when we have languages that can be statically type-checked. Using a statically type-checked specification language is even more advisable than using a statically type-checked programming language. It seems strange that some formulae might make sense "accidentally" — only for particular implementations of the specification or its parameters. So I shall pursue the "static type-checking" scenario, adding sharing to the framework.

There are two ways of adding sharing to the framework:

External sharing — sharing is resolved outwith the institutional notion of signature. For example, we could maintain a map from "external identifiers" to "internal names," the latter being names in an algebraic signature. This is (a bit) like the 1990 SML semantics, and was suggested in the algebraic semantics sketched for EML by Sannella and Tarlecki [1986].

Internal sharing — sharing is made part of signatures in the institution. For example, to handle type sharing, signatures might be given an equivalence relation on sorts.

The advantage of external sharing is that we stick to a familiar-looking institution with ordinary algebraic signatures. The considerable disadvantage is that we completely break the institution-independent framework: for example, the specification building operators of ASL would have to be lifted to operate on the "external" part of algebraic signatures. Much extra complexity is hidden in the translation function mapping the source language into the institutional semantics, and it would have to be defined afresh whenever we modify the source language.

The alternative, having sharing internal to signatures, means that we must modify the institution. After that, we retain the advantage of being able to apply the familiar institution-independent framework. Following this choice, signatures are *static typing environments* which contain the complete information needed to type-check terms and formulae. This means that $\mathbf{Sen}(\Sigma)$ is exactly the set of well-typed formulae in the abstract syntax over the signature Σ.

The advantage of an institutional semantics is an excellent motivation for following the internal sharing approach. There is still a question about how the sharing information is expressed. We could use an equivalence relation on sort names, or perhaps a class of models over some subsignature to force the interpretation of some of the symbols.[11]

[11] this last idea is reminiscent of hierarchical specifications, see Wirsing [1990].

Chapter 6 defines an institution $\mathcal{FPC}$ which expresses sharing by adding "symbolic type equations" to signatures. This allows signatures to remain close to the concrete syntax of the language, because names in a signature are also used as program-level identifiers. For $\mathcal{FPC}$, the sentence functor is defined directly upon the richer signatures. This approach takes advantage of having syntax to describe algebras: the symbolic type equations are composed of type-expressions in the underlying programming language.

5.11 Related Work

Researchers have studied numerous foundations for module parameterisation in specification languages [for brief surveys see Wirsing, 1990, Sannella et al., 1992] and also in programming languages [for references see Leroy, 1996b]. But there is hardly any work on modules for wide-spectrum languages which encompass both program and specification parameterisation. Exceptions include the original description of ASL+ [Sannella et al., 1992] and the languages EML [Kahrs et al., 1994] and SPECTRAL [Krieg-Brückner and Sannella, 1991], which each have Π-abstractions, but no λ-abstraction over specifications.

Here I shall mention some work related to ASL+ from the areas of type theory and specification; this is not an attempt at a full survey. Related work on programming languages (which relates more with the last two sections above) will be compared in Chapter 7.

Module algebra [Bergstra et al., 1990] Specifications in ASL+ are built using SBOs, which were formalized in Section 5.2 as n-ary operations in an algebraic signature. The *module algebra* of Bergstra et al. uses operators for building specifications in $\mathcal{FOL}$, specified as an abstract datatype. Rather than a sort for (the algebras in) each signature, there is a single sort M of modules, together with an operation $Sig(-) : M \to \mathsf{Sign}$. The abstract datatype has axioms for the laws that the SBOs must satisfy. In ASL+, such laws are instead expresssed via $\models^{sat}$. We could use the operations of module algebra as an alternative set of SBOs for $\lambda_{\mathsf{ASL+}}$, for the particular institution $\mathcal{FOL}$.

Type theoretic approaches Several researchers have tried to fit algebraic specification languages inside type theories; Luo [1993] gives a nice reformulation of ASL inside ECC, for example. Type theory appears promising for program development: specifications and programs can be written in the same language, and there are useful constructs for modularisation, including

λ-abstraction for parameterisation, Π-abstraction for specifying parameterisation programs which depend on their arguments, and Σ-types for building up programs. These last two features are exploited by Burstall and McKinna [1991] in the *deliverables* approach to program development; Reus and Streicher [1992] reformulated the axioms of module algebra in this setting.

But type theory is not (yet) a panacea; in particular, Σ-types do not provide the right explanation of modular programming (dependent records would be better) and user-level support for writing modules is only now being investigated [Courant, 1997b]. Moreover, the ASL+ approach works not in an obese type theory, but for an *arbitrary* programming and specification language, without adding anything new to them.

COLD-K [Jonkers, 1989] The specification language COLD-K has a parameterisation mechanism studied by Feijs [1989]. It is based on λ-calculus extended with a *conditional β-reduction rule*, called π:

$$(\lambda x \sqsubseteq R.M)N \twoheadrightarrow_{\pi} M[N/x] \qquad \textit{provided}\, N \sqsubseteq R$$

The term R acts as a so-called *parameter restriction*. The relation $\sqsubseteq$ is defined on terms of the calculus, extending reflexivity and assumptions pointwise to λ-terms. The proof system for $M \sqsubseteq N$ includes a contravariant rule for λ-terms. This gives a calculus of higher-order parameterisation which has similarities with the type-level of higher-order subtyping calculi mentioned in the last chapter (see Section 4.3.2), and hence with $\lambda_{\mathsf{ASL+}}$. The conditional rule for β-conversion corresponds to the usual check that an argument type $A \leq B$ in the rule for applying type operators $(\lambda\alpha \leq B.M)A$. A term cannot be written in the (full) type systems studied here unless it satisfies the parameter restriction.

To limit the terms that can be written in the COLD-K λ-calculus, Feijs gives a Curry type-assignment system, starting from an algebra of constants over a single sort and a pre-order on the sort. A model is constructed by extending the algebra with a "top" element of its domain, and interpreting Curry typed terms as functions, in the usual way.

This is close to the ideas behind rough typing and SBOs in ASL+, except that Feijs begins from a single type 0 for all specifications, rather than types $\mathsf{P}(\Sigma)$ for each Σ.[12] A single type for all specifications gives a more general notion of parameter: one can write functionals operating on specifications of arbitrary signature, for example. Arguably, such flexibility is not so useful when programming and specifying in-the-large, because we want modules

[12]He mentions the possibility for other sorts, but the suggestion is that these would be used for *syntactic* purposes: parameterised renamings and the like.

to have fixed input and output interfaces to comminicate requirements between separate parts of the development (the interfaces should be fixed, but arguments which are more general than the interfaces should be allowed). But this is a point of methodology rather than a criticism of the language.

Compared with ASL+, COLD-K has no Π-abstraction (and no level of programs).

Cengarle and Wirsing [1994] A similar specification language with higher-order parameterisation based on λ-calculus was introduced by Cengarle and Wirsing and studied in detail in Cengarle's thesis [1994]. The basic calculus is similar to COLD-K's λ-calculus, but instead of conditional β-reduction, Cengarle provides her language with a formal system for deriving a set of *requirements* which must be satisfied for any particular parameter. She also distinguishes a type of specifications 1 from the type of signatures 0, and allows parameterised signatures as well as parameterised specifications. Later, parameterised requirements enter the picture.

A crucial part of Cengarle's language is the use of the function $Sig(-)$ as part of the object language, rather than a meta-language entity (this idea is suggested in Wirsing [1990]). It means that the body of a parameterised specification can *compute* the desired output signature based on the signature of its parameter, which may vary. This is a flexible technique, but still requires the user to explicitly construct combinations of signatures — for a high-level language, we might prefer that pushouts or amalgamations were constructed automatically.

Apart from these differences, ASL+ stands apart from Cengarle's language because it includes Π-specifications and parameterisation on programs, as well as parameterised specifications, and is given a model-theoretic semantics rather than a presentation-level semantics (see the discussion in Chapter 1 on page 18).

5.12 Summary

This chapter described the idea behind ASL+ as a generic construction for building a modular programming and specification language from an arbitrary institution. An abstract version of ASL+ was defined which was intended to realise this goal, based on modifications to the original work by Sannella et al. [1992].

The abstract version of ASL+ was defined using $\lambda_{\mathsf{ASL+}}$, an extension of the type systems $\lambda_{\leq\{\}}$ and λ_{Power} of the previous chapters. The definition of $\lambda_{\mathsf{ASL+}}$ began with a rough type-checking system and a semantics based on the definitions given for λ_{Power}. The "full" type system, called the satisfaction system, was based on a consequence relation $\vdash^{sat}$ for proving satisfaction for simple specifications and programs built with PBOs and SBOs. All the results proved for λ_{Power} in Chapter 4 also hold for $\lambda_{\mathsf{ASL+}}$.

To define ASL+ itself, specific PBOs and SBOs were defined, for constructing programs and specifications based on an arbitrary institution. The syntax and semantics of these operators is based on Sannella and Tarlecki [1988a,b]. Then a proof system defining $\vdash^{sat}$ was sketched, demonstrating some old rules from Farrés-Casals [1992] and some new (but unexciting) rules; further research is needed here.

This abstract description of ASL+ can serve only as a kernel language; examples such as those in Chapter 2 need translation to be fully formalized. This is because of two problems described in Section 5.8: the *context problem* and the *sharing problem*. These problems are related, in fact, since we wish to find a useful form of context and at the same time explain sharing between the context and the body of parameterised program or specification.

These problems were explained and explored in the later part of this chapter. The exploration motivated a design decision. In Section 5.10 I described the idea of including symbolic type expressions in the signatures of an institution, so that the names of the signature can really be used as program-level identifiers, as in the examples in Chapter 2. This idea is taken up in Chapter 6.

6 The Institution $\mathcal{FPC}$

This chapter defines an institution $\mathcal{FPC}$, complete with a syntax for signatures and algebras. It is based on a small functional programming language together with a suitable program logic. The functional language is FPC, which has partial recursive functions and recursive types. The program logic is a version of Higher Order Logic, extended with constructs for reasoning about the types of FPC.

In the next chapter, the institution $\mathcal{FPC}$ is used to define ASL+$_{\text{FPC}}$, a concrete version of ASL+.

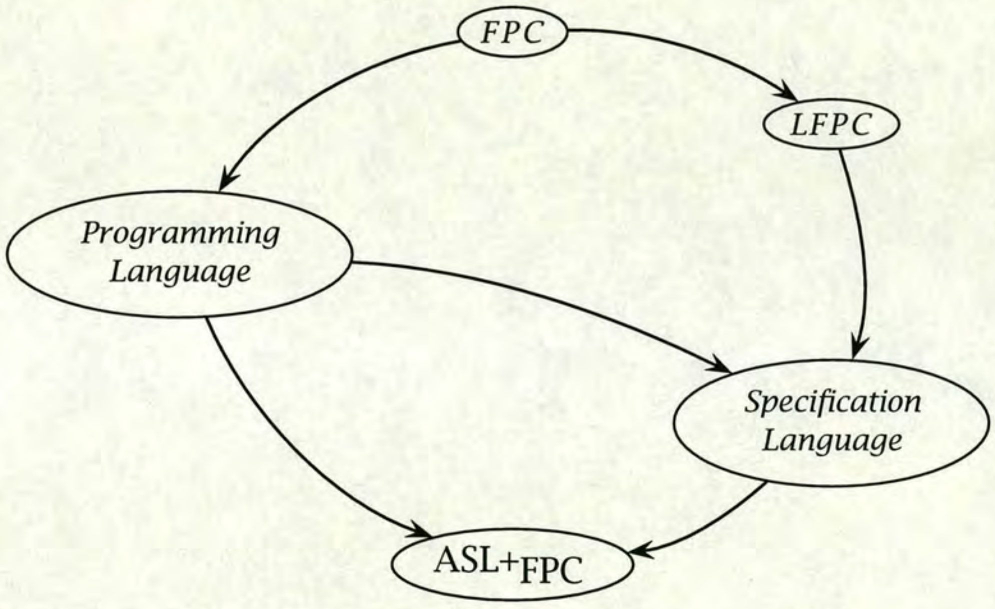
6.1 Overview

THE LANGUAGE ASL+ was conceived as a construction to build a modular specification and programming language from a core-level programming language and a corresponding program logic. The construction is supposed to work for any programming language and program logic with an institutional semantics.

Institution-independent generalities are all very well, but to illustrate the use of ASL+, we present examples using specific instances. A good way to be encouraged that generalities work is to carefully examine particular instances, spelling out low-level details it is tempting to omit.

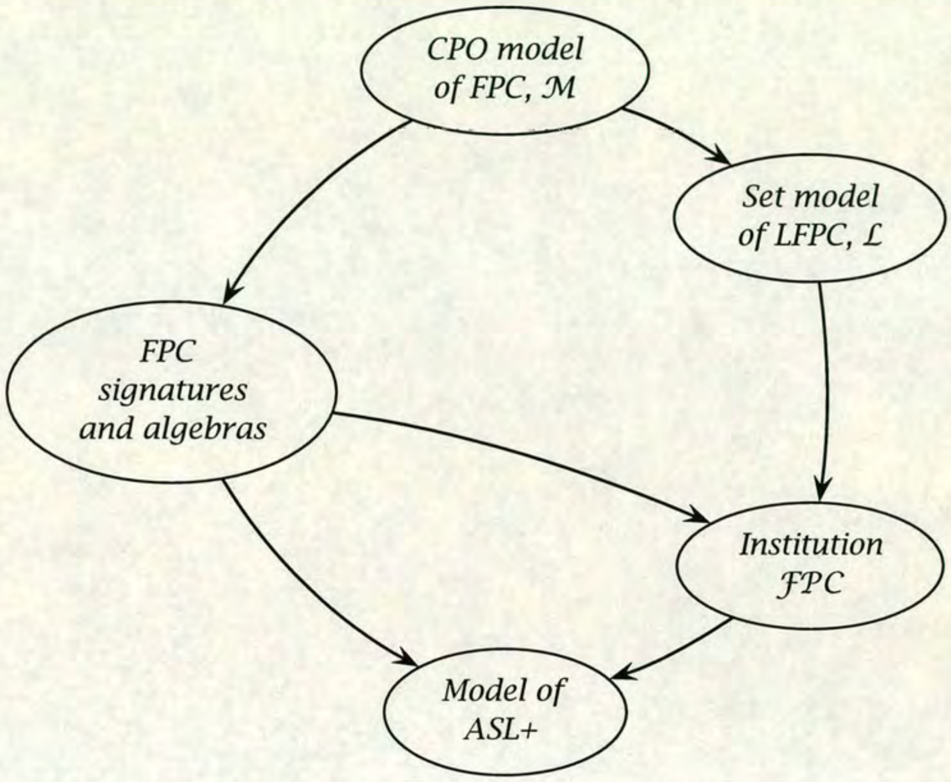
This chapter and the following one present my attempt to examine one particular instance of ASL+. Rather than use a programming language contrived from the simplest institution $\mathcal{EQ}$, I shall use a language which more closely resembles a modern functional programming language such as Standard ML, together with a suitable logic for writing specifications. It should be able to formalize the examples earlier in the thesis completely. This means in the core language, dealing with higher-order types, and both recursive functions and recursive types; in the module language, it means handling sharing and the propagation of type equalities.

As a picture, the syntax fits together like this:



The programming language is built upon FPC, a minimal λ-calculus with recursive types. Programs are groups of declarations of FPC types and terms. Signatures are the "types" of programs; declarations of type names and value names with types. The program logic, dubbed LFPC, is a version of Higher Order Logic (HOL) extended with LCF-like constructs for reasoning about the types of FPC. Terms of FPC are also terms in LFPC. The propositions of LFPC, together with the syntax for signatures and the SBOs of ASL, give a specification language. Combining the programming and specification languages and adding the ASL+ λ-calculus for higher-order modules, we finally arrive at ASL+_{FPC} .

For each piece of the syntax, there is a corresponding semantics:


CPO model of FPC, $\mathcal{M}$
Set model of LFPC, $\mathcal{L}$
FPC signatures and algebras
Institution $\mathcal{FPC}$
Model of ASL+

The typed λ-calculus FPC has a standard CPO model $\mathcal{M}$ in a category which allows the solution of recursive domain equations. The FPC signatures and algebras are the denotations of signature expressions and programs in the programming language. The logic LFPC has a set-theoretic model $\mathcal{L}$ which interprets the predicate $\sqsubseteq$ on an FPC type as the CPO order relation on its domain in $\mathcal{M}$. (The type itself is modelled as the underlying set of the domain.) Formulae from LFPC are combined with algebras and signatures in the institution for FPC, $\mathcal{FPC}$. Finally we can give a semantics for $\mathrm{ASL+}_{\mathrm{FPC}}$, in a similar form to the semantics defined in earlier chapters.

This chapter contains the "core" language part of this construction, which defines the institution $\mathcal{FPC}$ and the syntax for signatures and algebras. Section 6.2 introduces the syntax for FPC; Section 6.3 defines its semantics. Section 6.4 introduces the syntax for LFPC, and Section 6.5 defines its semantics. Section 6.6 sketches a proof system for the logic. The institution $\mathcal{FPC}$ is defined in Section 6.7, and a syntax for programs and signatures follows in Section 6.8. More details of FPC appear in Appendix A, including definitions of standard types and a presentation of the LFPC proof system.

Section 6.9 discusses the design of $\mathcal{FPC}$ and compares it with related work. Section 6.10 concludes with a summary. The study of the module calculus proper begins in Chapter 7.

6.2 The Language FPC

The language FPC (Fixed Point Calculus) is an extension of the simply-typed lambda calculus with product types, $s \times t$, sum types $s + t$, and recursive types $\mu a.t$ [Plotkin, 1985, Gunter, 1992].

With these type constructors no base types are necessary because familiar datatypes can be built-up beginning from the empty type $\mu a.a$. For each function type $s \rightharpoonup p$, we can define a fixed point operator so that we can program with recursive functions. The expressiveness of FPC is well-known; for reference, Appendix A has some more details of how the familiar datatypes and fixed point operators can be defined.

In practice, of course, the full type expressions for familiar datatypes are too cumbersome to write out every time, and we need to declare type definitions for abbreviation. For the same reason, we need to declare terms, such as the fixed point operator. Declarations are added at the level of programs (rather than expressions); to allow for them we add type and term constants to the language of expressions.

This section reviews the syntax of FPC. The presentation given here differs from those cited above, because it is factored by a signature to specify type and term constants.

6.2.1 FPC types and signatures

Let *TyVar* and *TyConst* be disjoint countable sets, the *type variables* and the *type constants*. I use $a, \ldots$ to range over *TyVar* and $c, d, \ldots$ to range over *TyConst*. The set of FPC types, ranged over by t, is generated by the grammar:

$$t \ ::= \ c \ \mid \ a \ \mid \ t \rightharpoonup t \ \mid \ t \times t \ \mid \ t + t \ \mid \ \mu a.t$$

The free and bound variables of a type are defined as usual, and α-convertible types are considered syntactically identical. Substitution of the type s for the type variable a in the type t is written $[s/a]t$.

Given a subset $Ty \subseteq TyConst$, we write $ProgTypes(Ty)$ for the set of closed types containing constants only from Ty.

The syntax of FPC is parameterised on a notion of signature, which is equipped with *sharing equations* for type constants.[1] Let *TmConst* be another countable set, the *term constants*. I use $v, \ldots$ to range over *TmConst*. Occasionally I use ν, ν' to range over $TyConst \cup TmConst$.

[1] In place of the usual algebraic terminology of "sorts" and "operators," I will oten speak of "type constants" and "term constants".

Definition 6.1 (FPC Signatures).
An *FPC signature* Σ is a triple $(Ty^{\Sigma}, Sh^{\Sigma}, Tm^{\Sigma})$ where

- $Ty^{\Sigma} \subseteq \mathit{TyConst}$ is a set of type constants,
- $Sh^{\Sigma} : Ty^{\Sigma} \to \mathit{Fin}(\mathit{ProgTypes}(Ty^{\Sigma}))$ is an assignment of finite sets of types to type constants,
- $Tm^{\Sigma} : \mathit{TmConst} \rightharpoonup \mathit{ProgTypes}(Ty^{\Sigma})$ is a partial assignment of types to term constants. □

The set of type constants Ty^{Σ} determines the set of Σ-types, $\mathit{ProgTypes}(Ty^{\Sigma})$. For simplicity, signatures do not allow overloading, so a partial function Tm^{Σ} gives the types of term constants; the set of term constants defined by Σ is $\mathit{Dom}(Tm^{\Sigma})$.

The component Sh^{Σ} expresses sharing between types, or type definitions so a type constant can abbreviate a complicated type-expression. (See Section 5.10 for motivation behind adding type equations to signatures.)

The equations given by Sh^{Σ} are asymmetric. In principle the sharing equations of a signature should be unifiable and have a solution as a set of type assignments for the type constants of the signature, in other words, there should be a partial function from Ty^{Σ} to $\mathit{ProgTypes}(Ty^{\Sigma})$. Instead, at this stage a function into $\mathit{Fin}(\mathit{ProgTypes}(Ty^{\Sigma}))$ allows signatures to be put together easily. No restrictions are put on Sh^{Σ} to prevent inconsistent equalities between types; none are necessary for the following definitions to make sense, and such "monster" signatures[2] are prevented in the concrete syntax. However, inconsistent sharing equations can arise when type-checking the module language (see the discussion on page 258).

Definition 6.2 (Type equality generated by a signature).
A signature $\Sigma = (Ty^{\Sigma}, Sh^{\Sigma}, Tm^{\Sigma})$ gives rise to an equality relation on types, written $=_{\Sigma}$, which is defined to be the least equivalence relation generated by the set $\{ c = t \mid t \in Sh^{\Sigma}(c), c \in Ty^{\Sigma} \}$ and compatible with the type formers.

(Notice that $=_{\Sigma}$ does not identify a recursive type with its unfolding — the equality is intended to identify types which have the same elements, but a recursive type and its unfolding are only isomorphic in the syntax of terms considered below.)

[2]This is the fearsome terminology used in SML.

Definition 6.1 defines a rather syntactic form of signature; we would expect the two signatures

```
sig                    sig
   type c                 type c
   type d                 type d
   sharing c = d          sharing d = c
end                    end
```

to be essentially equivalent.[3] Signatures use Sh^{Σ}, rather than the equality relation $=_{\Sigma}$ directly, because it allows a practical implementation to recover syntactic representations of signatures. (Researchers criticized the 1990 definition of SML [Milner et al., 1990] for not allowing this.) Semantically we consider signatures modulo an equivalence relation, given in Definition 6.9.

Notation 6.3. Let $\Sigma = (Ty^{\Sigma}, Sh^{\Sigma}, Tm^{\Sigma})$ be a signature. I will use several shorthand notations:

Notation	Meaning
$\mathit{ProgTypes}(\Sigma)$	$\mathit{ProgTypes}(Ty^{\Sigma})$
$c \in \Sigma$	$c \in Ty^{\Sigma}$
$c := t \in \Sigma$	$t \in Sh^{\Sigma}(c)$
$v \in \Sigma$	$v \in Dom(Tm^{\Sigma})$
$v : t \in \Sigma$	$Tm^{\Sigma}(v) = t$
$\Sigma \cup \{ c \}$	$(Ty^{\Sigma} \cup \{ c \}, Sh^{\Sigma}, Tm^{\Sigma})$
$\Sigma \cup \{ c := t \}$	$(Ty^{\Sigma} \cup \{ c \}, Sh^{\Sigma}[c \mapsto \left\{ Sh^{\Sigma}(c) \cup \{ t \} \right\}], Tm^{\Sigma})$
$\Sigma \cup \{ v : t \}$	$(Ty^{\Sigma}, Sh^{\Sigma}, Tm^{\Sigma}[v \mapsto t])$
$\varnothing$	$(\{ \}, \{ \}, \{ \})$

In the last case $\varnothing$ is the empty signature.

This notation is used for triples (Ty, Sh, Tm) which are not necessarily proper signatures, perhaps because the domain of Sh is not Ty, or because part of the range of Sh or Tm lies outside $\mathit{ProgTypes}(Ty)$. To draw attention to this, we may call such a triple a *pre-signature*. All the notations above make sense for pre-signatures.

6.2.2 FPC terms and type-checking

Let TmVar be a countable set of *term variables*, disjoint from $\mathit{TmConst}$. I shall use $x, y, \ldots$ to range over TmVar. The set of FPC terms is given by the grammar:

[3] The syntax used here is defined formally in Section 6.8.2.

$$
\begin{array}{rcl}
e & ::= & v \mid x \\
& \mid & \mathbf{fun}\,(x:t).\,e \mid e\,e \\
& \mid & \langle e,e\rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \\
& \mid & \mathbf{inl}_{t+t}(e) \mid \mathbf{inr}_{t+t}(e) \\
& & \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e\ \mathbf{or}\ \mathbf{inr}(x) \Rightarrow e \\
& \mid & \mathbf{intro}_{\mu a.t}(e) \mid \mathbf{elim}(e)
\end{array}
$$

The free and bound variables of a term are defined as usual α-convertible terms are identified. Substitution for terms is written $e[f/x]$.

Given a signature Σ, the set of Σ-terms, $\mathit{ProgTerms}(\Sigma)$, is the set of closed terms which contain only term constants $v \in \Sigma$ and types in $\mathit{ProgTypes}(\Sigma)$.

FPC is a typed language, and only typable terms are assigned a meaning in the semantics. A term is typed with respect to a signature Σ and a context G of assumptions $x : t$ about the types of free variables. As usual, typing contexts must not contain repeated declarations for the same variable. The typing judgement for FPC is:

$G \rhd^{\Sigma} e : t$ "term e has type t in context G, in signature Σ"

It is defined in Figure 6.1.

Typing in FPC has some good properties.

Proposition 6.4 (FPC typing has good properties).

1. *(Closure).*
   If $G \rhd^{\Sigma} e : t$, then $FV(e) \subseteq Dom(G)$ and $t \in \mathit{ProgTypes}(\Sigma)$.
   Moreover, if a term constant v appears in e, then $v \in \Sigma$.

2. *(Type unicity).*
   If $G \rhd^{\Sigma} e : s$ and $G \rhd^{\Sigma} e : t$ then $s =_{\Sigma} t$.

Proof Standard. □

Using Proposition 6.4(1), if $\rhd^{\Sigma} e : t$, then $e \in \mathit{ProgTerms}(\Sigma)$.

6.2.3 Changing signature

A signature morphism is a consistent renaming of the type and term constants in a signature, which preserves sharing.

Definition 6.5 (FPC Signature Morphisms).
A *signature morphism* $\sigma : \Sigma \to \Sigma'$ is a pair $(Ty^{\sigma}, Tm^{\sigma})$ where

- $Ty^{\sigma} : Ty^{\Sigma} \to Ty^{\Sigma'}$

$$\frac{v : t \in \Sigma}{G \rhd^{\Sigma} v : t} \qquad \frac{G \rhd^{\Sigma} e_1 : s}{G \rhd^{\Sigma} \mathbf{inl}_{s+t}(e_1) : s + t}$$

$$\frac{t \in \mathit{ProgTypes}(\Sigma)}{G, x : t, G' \rhd^{\Sigma} x : t} \qquad \frac{G \rhd^{\Sigma} e_2 : t}{G \rhd^{\Sigma} \mathbf{inr}_{s+t}(e_2) : s + t}$$

$$\frac{G, x : s \rhd^{\Sigma} e : t}{G \rhd^{\Sigma} \mathbf{fun}\,(x : s).\, e : s \rightharpoonup t}$$

$$\frac{G \rhd^{\Sigma} e_1 : s \rightharpoonup t \qquad G \rhd^{\Sigma} e_2 : s}{G \rhd^{\Sigma} e_1\, e_2 : t} \qquad \frac{\begin{array}{c} G \rhd^{\Sigma} e : s_1 + s_2 \\ G, x : s_1 \rhd^{\Sigma} e_1 : t \\ G, y : s_2 \rhd^{\Sigma} e_2 : t \end{array}}{G \rhd^{\Sigma} \mathbf{case}\, e \begin{array}{ll} \mathbf{of} & \mathbf{inl}(x) \Rightarrow e_1 \\ \mathbf{or} & \mathbf{inr}(y) \Rightarrow e_2 \end{array} : t}$$

$$\frac{G \rhd^{\Sigma} e_1 : s \qquad G \rhd^{\Sigma} e_2 : t}{G \rhd^{\Sigma} \langle e_1, e_2 \rangle : s \times t} \qquad \frac{G \rhd^{\Sigma} e : [\mu a.t/a]t}{G \rhd^{\Sigma} \mathbf{intro}_{\mu a.t}(e) : \mu a.t}$$

$$\frac{G \rhd^{\Sigma} e : s \times t}{G \rhd^{\Sigma} \mathbf{fst}(e) : s} \qquad \frac{G \rhd^{\Sigma} e : \mu a.t}{G \rhd^{\Sigma} \mathbf{elim}(e) : [\mu a.t/a]t}$$

$$\frac{G \rhd^{\Sigma} e : s \times t}{G \rhd^{\Sigma} \mathbf{snd}(e) : t} \qquad \frac{G \rhd^{\Sigma} e : s \qquad s =_{\Sigma} t}{G \rhd^{\Sigma} e : t}$$

Figure 6.1: FPC type-checking

- $Tm^\sigma : Dom(Tm^\Sigma) \to Dom(Tm^{\Sigma'})$

are functions such that

$$c := t \in \Sigma \quad \Rightarrow \quad \sigma(c) =_{\Sigma'} \sigma(t)$$

$$v : t \in \Sigma \quad \Rightarrow \quad Tm^{\Sigma'}(\sigma(v)) =_{\Sigma'} \sigma(t)$$

for all $c, v \in \Sigma$. As usual, σ also stands for the homomorphic extension of Ty^σ to types or for the extension of Tm^σ and Ty^σ to terms. □

A signature morphism $\sigma : \Sigma \to \Sigma'$ translates Σ types and terms to Σ' types and terms, preserving the type equality of Σ. The translation also preserves typing; the next proposition establishes this. If G is a context, $\sigma(G)$ is the context obtained by replacing each declaration $x_i : t_i$ in G with $x_i : \sigma(t_i)$.

Proposition 6.6 (FPC typing is preserved by signature change).
Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism.

1. $t \in \mathit{ProgTypes}(\Sigma) \quad \Rightarrow \quad \sigma(t) \in \mathit{ProgTypes}(\Sigma')$.
2. $s =_\Sigma t \quad \Rightarrow \quad \sigma(s) =_{\Sigma'} \sigma(t)$.
3. $G \rhd^\Sigma e : t \quad \Rightarrow \quad \sigma(G) \rhd^{\Sigma'} \sigma(e) : \sigma(t)$.
4. $e \in \mathit{ProgTerms}(\Sigma) \quad \Rightarrow \quad \sigma(e) \in \mathit{ProgTerms}(\Sigma')$.

Proof

1. Immediate from definition of signature morphism.
2. By induction on the generation of the equality $s =_\Sigma t$. In the base case where $c := t \in \Sigma$, it follows immediately from the definition that $\sigma(c) =_{\Sigma'} \sigma(t)$.
3. Easy induction on the derivation of $G \rhd^\Sigma e : t$.
4. Immediate from part 3. □

A special case of signature morphism is the inclusion between a signature Σ and a richer one Σ' having more constants or equalities.

Definition 6.7 (FPC subsignatures and inclusions).
A signature Σ is a *subsignature* of Σ', written $\Sigma \subseteq \Sigma'$, if

- $Ty^{\Sigma} \subseteq Ty^{\Sigma'}$,
- $c := t \in \Sigma \quad \Rightarrow \quad c =_{\Sigma'} t$, and
- $v : t \in \Sigma \quad \Rightarrow \quad Tm^{\Sigma'}(v) =_{\Sigma'} t$.

When $\Sigma \subseteq \Sigma'$, there is a canonical signature morphism $\iota_{\Sigma,\Sigma'} : \Sigma \hookrightarrow \Sigma'$, the *inclusion of Σ in Σ'*, composed of the evident inclusions.

Example 6.8. Define three signatures by:

$\Sigma_1 =_{\text{def}}$	$\Sigma_2 =_{\text{def}}$	$\Sigma_3 =_{\text{def}}$
sig	**sig**	**sig**
type c	**type** c	**type** c
val $v : (c \times bool)$	**type** d	**type** d
$\times(c \times bool)$	**sharing** $d = c \times bool$	**val** $v : (c \times bool) \times d$
end	**val** $v : d \times d$	**end**
	end	

Then $\Sigma_1 \subseteq \Sigma_2$ and $\Sigma_3 \subseteq \Sigma_2$ but Σ_1 and Σ_3 are unrelated. □

The subsignature relation is reflexive and transitive. It generates an equivalence relation on signatures which is the intended equality between signatures. Signature morphisms also have a intended equality.

Definition 6.9 (FPC signature and signature morphism equality).

- Let Σ, Σ' be signatures. Then $\Sigma = \Sigma'$ iff $\Sigma \subseteq \Sigma' \ \bigwedge \ \Sigma' \subseteq \Sigma$.
- Suppose $\sigma_1, \sigma_2 : \Sigma \to \Sigma'$. Then $\sigma_1 = \sigma_2$ iff

$$\forall c \in \Sigma.\ \sigma_1(c) =_{\Sigma'} \sigma_2(c) \quad \bigwedge \quad \forall v \in \Sigma.\ \sigma_1(v) = \sigma_2(v).$$

6.3 Fixed Point Semantics of FPC

This section recalls the standard domain construction in Gunter and Scott [1990], which gives FPC types a fixed-point semantics using a *universal domain $\mathcal{U}$*. Then an environment model is defined for FPC.

6.3.1 Universal domain

A universal domain can give a semantics for recursive types, using the fixed point theorem for CPOs to solve recursive domain equations. To define the semantics ofFPC, it is enough to rely on abstract properties of the universal domain without worrying about its concrete construction.

The definitions in this section are standard.

CPOs. A *cpo D* is a partially ordered set in which every directed subset has a least upper bound. The underlying set of D is written as $|D|$. A *pointed cpo*, or *cppo*, is a cpo which has a least element. The *lift* of a cpo D is the cppo $D_\perp$ whose underlying set is $D \cup \{ \perp \}$ for $\perp \notin D$, and whose partial order is that of D extended by making $\perp$ least. I write $\mathsf{up} : D \to D_\perp$ for the injection function. Given a cppo D we can form the cpo $D_\downarrow$ which has the least element removed. A function $f : D \to E$ between cpos is *continuous* if it is monotone and preserves least upper bounds of directed subsets.

CPO constructions. The *cartesian product* of two cpos D and E is $D \times E$ which has the set of all pairs (d, e) with $d \in D$, $e \in E$ as underlying set, ordered componentwise. If D and E are pointed, their *smash product* is the cppo $D \otimes E$, which is defined as $(D_\downarrow \times E_\downarrow)_\perp$. The function $\mathsf{smash} : D \times E \to D \otimes E$ is defined by

$$\mathsf{smash}(d, e) = \begin{cases} (d, e) & \text{if } d \neq \perp \text{ and } e \neq \perp, \\ \perp_{D \otimes E} & \text{otherwise.} \end{cases}$$

I write the projection functions (strict on $D \otimes E$) as fst and snd.

The *disjoint union* of cpos D and E is the cpo $D \uplus E$ which has as underlying set $\{ (D \times \{ 0 \}) \cup (E \times \{ 1 \}) \}$, where $(d, 0) \sqsubseteq_{D \uplus E} (d', 0)$ if $d \sqsubseteq_D d'$, and similarly for elements from E. If D and E are cppos, their *coalesced sum* is the cppo $D \oplus E$, defined as $(D_\downarrow \uplus E_\downarrow)_\perp$. The functions $\mathsf{inl} : D \to D \oplus E$ and $\mathsf{inr} : E \to D \oplus E$ are defined by:

$$\mathsf{inl}(d) = \begin{cases} (d, 0) & \text{if } d \neq \perp, \\ \perp_{D \oplus E} & \text{otherwise.} \end{cases} \qquad \mathsf{inr}(e) = \begin{cases} (e, 1) & \text{if } e \neq \perp, \\ \perp_{D \oplus E} & \text{otherwise.} \end{cases}$$

For every pair of functions $f : D \to F$ and $g : E \to F$, where F is a cppo, the sum eliminator $[f, g] : D \oplus E \to F$ is defined by:

$$[f, g](x) = \begin{cases} f(d) & \text{if } x = (d, 0), \\ g(e) & \text{if } x = (e, 1), \\ \perp_F & \text{otherwise.} \end{cases}$$

The *continuous function space* of two cpos D and E is $[D \to E]$, the cpo of continuous functions from D to E ordered pointwise. Given a continuous function $f : [D \to E]$, strict f is the element of $[D_\perp \to E_\perp]$ defined by:

$$\mathsf{strict}(f)(d) = \begin{cases} \perp_E & \text{if } x = \perp_D, \\ f(x) & \text{otherwise.} \end{cases}$$

We assume some category **Dom** whose objects are *domains*, which are cpos of some kind, and whose morphisms are continuous functions. The precise details of **Dom** are not important to the construction; it is enough to know that it is closed under the above domain constructors and that it possesses a *universal domain*, $\mathcal{U}$, in the following sense. Given a domain D, we say that E is a sub-domain of D, written $E \lhd D$, if $|E| \subseteq |D|$ and there is a morphism $p : D \to E$ such that $p \circ p = p$ and $p(x) \sqsubseteq_D x$ for each $x \in D$. We require that $\mathcal{U}$ has a isomorphic copy of every domain D as a subdomain, and that the collection of all subdomains of $\mathcal{U}$ also forms a domain; thus there is a special subdomain $\mathcal{P}U \lhd \mathcal{U}$ which is isomorphic to the domain of subdomains of $\mathcal{U}$. Let φ stand for the isomorphism, so $\varphi(D) \sqsubseteq \varphi(E)$ iff $D \lhd E$ for $D, E \lhd \mathcal{U}$.

To solve domain equations, $\mathcal{U}$ should have continuous functions corresponding to the domain constructors. A continuous function $R : \mathcal{P}U \to \mathcal{P}U$ *represents* a domain constructor $F : \mathbf{Dom} \to \mathbf{Dom}$ if $F(D) \cong \varphi^{-1}(R(\varphi(D_1)))$ where $D \cong D_1 \lhd U$. For FPC, we need continuous functions on $\mathcal{P}U$ as follows:

- $U_{\rightarrow}$ which represents $(D, E) \mapsto [D \to E]_\perp$;
- $U_\times$ which represents the smash product;
- U_+ which represents the coalesced sum.

From now on, we assume a fixed category **Dom** with a universal domain $\mathcal{U}$ satisfying these properties.

6.3.2 *Interpretation of FPC*

The interpretation of types $\mathcal{M}[\![t]\!]_\iota$ is defined relative to a *type environment* ι which is a partial function that assigns pointed domains[4] to type variables and type constants.

[4]precisely, elements $\varphi(D)$ of $\mathcal{P}U$ corresponding to pointed domains D.

Definition 6.10 (Interpretation of FPC types).
Let t be a type expression and ι a type environment which is defined on the free variables and type constants in t. The interpretation of t is defined by induction on its structure:

- $\mathcal{M}[\![a]\!]_\iota = \iota(a)$
- $\mathcal{M}[\![c]\!]_\iota = \iota(c)$
- $\mathcal{M}[\![s \rightarrow t]\!]_\iota = U_{\rightarrow}(\mathcal{M}[\![s]\!]_\iota, \mathcal{M}[\![t]\!]_\iota)$
- $\mathcal{M}[\![s \times t]\!]_\iota = U_{\times}(\mathcal{M}[\![s]\!]_\iota, \mathcal{M}[\![t]\!]_\iota)$
- $\mathcal{M}[\![s + t]\!]_\iota = U_{+}(\mathcal{M}[\![s]\!]_\iota, \mathcal{M}[\![t]\!]_\iota)$
- $\mathcal{M}[\![\mu a.t]\!]_\iota = \mathsf{fix}(D \mapsto \mathcal{M}[\![t]\!]_{\iota[a \mapsto D]})$ □

To interpret terms we need both a type environment ι and a (term) *environment ρ*, which is a partial function from $\mathit{TmVar} \bigcup \mathit{TmConst}$ to elements of domains. The two environments must be consistent with the signature and context. Because types occurring in well-typed terms are always closed, the type environment only needs to be defined on type constants from the signature.

Definition 6.11 (FPC Environments for signatures and contexts).
Let Σ be a signature.

1. The type environment ι is a *Σ-type environment* if $Dom(\iota) = Ty^\Sigma$ and
$$c := t \in \Sigma \quad \Rightarrow \quad \iota(c) = \mathcal{M}[\![t]\!]_\iota,$$
for all $c \in \mathit{TyConst}$.

2. The term environment ρ is a *(G, Σ, ι)-FPC environment* if $Dom(\rho) = Tm^\Sigma \cup Dom(G)$ and
$$v : t \in \Sigma \quad \Rightarrow \quad \rho(v) \in \mathcal{M}[\![t]\!]_\iota,$$
$$x : t \in G \quad \Rightarrow \quad \rho(x) \in \mathcal{M}[\![t]\!]_\iota,$$
for all $v \in \mathit{TmConst}$ and $x \in \mathit{TmVar}$.
If G is the empty context, we speak of a (Σ, ι)-FPC environment.

Given a Σ-type environment ι, Definition 6.10 assigns a semantic domain $\mathcal{M}[\![t]\!]_\iota$ to every type $t \in \mathit{ProgTypes}(\Sigma)$. Alpha-convertible types have the same denotation, as do any pair of types equal in Σ.

Fact 6.12. If ι is a Σ-type environment, then $s =_\Sigma t$ implies $\mathcal{M}[\![s]\!]_\iota = \mathcal{M}[\![t]\!]_\iota$.

Given a Σ-type environment ι and a (G, Σ, ι)-FPC environment ρ, Definition 6.13 below assigns a meaning $\mathcal{M}[\![G \rhd^\Sigma e : t]\!]_{\iota\rho}$ to terms e such that $G \rhd^\Sigma e : t$. The interpretation of e is an element of a domain D in **Dom** with $D \cong \mathcal{M}[\![t]\!]_\iota$. For clarity, we elide the isomorphisms between domains D and elements of $\mathcal{P}U$ in the definition.

Definition 6.13 (Intepretation of FPC terms).
Suppose $G \rhd^\Sigma e : t$. Let ι be a Σ-type environment and ρ be a (G, Σ, ι)-FPC environment. The interpretation of e in ι, ρ is defined by induction on the typing derivation:

- $\mathcal{M}[\![G \rhd^\Sigma v : t]\!]_{\iota\rho} = \rho(v)$
- $\mathcal{M}[\![G \rhd^\Sigma x : t]\!]_{\iota\rho} = \rho(x)$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{fun}\,(x : s).e : s \,{-}\rangle\, t]\!]_{\iota\rho} = \mathsf{up}(\mathsf{strict}(f))$ where f is defined by
$$f(d) = \mathcal{M}[\![G, x : s \rhd^\Sigma e : t]\!]_{\iota(\rho[x \mapsto d])}$$
for all $d \in \mathcal{M}[\![s]\!]_\iota$
- $\mathcal{M}[\![G \rhd^\Sigma e_1\, e_2 : t]\!]_{\iota\rho} = \mathcal{M}[\![G \rhd^\Sigma e_1 : s \,{-}\rangle\, t]\!]_{\iota\rho}\,(\mathcal{M}[\![G \rhd^\Sigma e_2 : s]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \langle e_1, e_2\rangle : s \times t]\!]_{\iota\rho} =$
$\mathsf{smash}(\mathcal{M}[\![G \rhd^\Sigma e_1 : s]\!]_{\iota\rho}, \mathcal{M}[\![G \rhd^\Sigma e_2 : t]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{fst}(e) : s]\!]_{\iota\rho} = \mathsf{fst}(\mathcal{M}[\![G \rhd^\Sigma e : s \times t]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{snd}(e) : t]\!]_{\iota\rho} = \mathsf{snd}(\mathcal{M}[\![G \rhd^\Sigma e : s \times t]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{inl}_{s+t}(e_1) : s + t]\!]_{\iota\rho} = \mathsf{inl}(\mathcal{M}[\![G \rhd^\Sigma e_1 : s]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{inr}_{s+t}(e_2) : s + t]\!]_{\iota\rho} = \mathsf{inr}(\mathcal{M}[\![G \rhd^\Sigma e_2 : t]\!]_{\iota\rho})$
- $\mathcal{M}[\![G \rhd^\Sigma \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2 \ :\ t]\!]_{\iota\rho} =$
$[f, g](\mathcal{M}[\![G \rhd^\Sigma e : s_1 + s_2]\!]_{\iota\rho})$
where f and g are defined by:
$$f(d) = \mathcal{M}[\![G, x : s_1 \rhd^\Sigma e_1 : t]\!]_{\iota(\rho[x \mapsto d])}$$
$$g(e) = \mathcal{M}[\![G, y : s_2 \rhd^\Sigma e_2 : t]\!]_{\iota(\rho[y \mapsto e])}$$
for all $d \in \mathcal{M}[\![s_1]\!]_\iota$, $e \in \mathcal{M}[\![s_2]\!]_\iota$

- $\mathcal{M}[\![G \rhd^{\Sigma} \mathbf{intro}_{\mu a.t}(e) : \mu a.t]\!]_{\iota\rho} = \mathcal{M}[\![G \rhd^{\Sigma} e : [\mu a.t/a]t]\!]_{\iota\rho}$
- $\mathcal{M}[\![G \rhd^{\Sigma} \mathbf{elim}(e) : [\mu a.t/a]t]\!]_{\iota\rho} = \mathcal{M}[\![G \rhd^{\Sigma} e : \mu a.t]\!]_{\iota\rho}$
- $\mathcal{M}[\![G \rhd^{\Sigma} e : t]\!]_{\iota\rho} = \mathcal{M}[\![G \rhd^{\Sigma} e : s]\!]_{\iota\rho}$ (where $s =_{\Sigma} t$). □

Alpha-convertible terms have the same denotation. Because typing derivations for any statement $G \rhd^{\Sigma} e : t$ differ only in the places that the type equality rule appears, and this rule is ignored in the interpretation function (last case above), the interpretation of $G \rhd^{\Sigma} e : t$ is independent of the typing derivation used to define it. Furthermore, by the unicity of FPC typing, the choice of type t cannot affect the interpretation of e. By these observations we have the following fact.

Fact 6.14. The interpretation of any typable term e in context G is unique, for a given choice of environments ι, ρ.

Section 6.7 extends this fine-grained semantics of types and terms to a notion of algebra for FPC. To end this section, we show that the interpretation of types and terms is unaffected by the choice of names for constants.

Definition 6.15 (Environment reducts).
Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Suppose ι is a Σ'-type environment and ρ is a $(\sigma(G), \Sigma', \iota)$-FPC environment. We define the *reduct of ι by σ*, written $\iota|_{\sigma}$ and the *reduct of ρ by σ*, written $\rho|_{\sigma}$, by:

$$\begin{array}{rcl} \iota|_{\sigma}(c) & = & \begin{cases} \iota(\sigma(c)) & \text{for all } c \in \Sigma, \\ \text{undefined} & \text{otherwise.} \end{cases} \\ \iota|_{\sigma}(a) & = & \iota(a) \\ \\ \rho|_{\sigma}(v) & = & \begin{cases} \rho(\sigma(v)) & \text{for all } v \in \Sigma, \\ \text{undefined} & \text{otherwise.} \end{cases} \\ \rho|_{\sigma}(x) & = & \rho(x) \end{array}$$

for all c, a, v, x. It follows directly that $\iota|_{\sigma}$ is a Σ-type environment and $\rho|_{\sigma}$ is a $(G, \Sigma, \iota|_{\sigma})$-FPC environment. □

Proposition 6.16 (FPC meaning is preserved by signature change).
Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Suppose ι is a Σ'-type environment and ρ is a $(\sigma(G), \Sigma', \iota)$-FPC environment. Then

- $\mathcal{M}[\![t]\!]_{\iota|_{\sigma}} = \mathcal{M}[\![\sigma(t)]\!]_{\iota}$
- $\mathcal{M}[\![G \rhd^{\Sigma} e : t]\!]_{\iota|_{\sigma}\rho|_{\sigma}} = \mathcal{M}[\![\sigma(G) \rhd^{\Sigma'} \sigma(e) : \sigma(t)]\!]_{\iota\rho}$

Proof By induction on the definition of $\mathcal{M}[\![-]\!]$, using Proposition 6.6. □

$$\frac{\tau \in \mathit{LogTypes}(\Sigma)}{G, z:\tau, G' \rhd^{\Sigma} z:\tau} \qquad \frac{G \rhd^{\Sigma} h : \tau \to \tau' \qquad G \rhd^{\Sigma} h' : \tau}{G \rhd^{\Sigma} h\,(h') : \tau'}$$

$$\frac{G \rhd^{\Sigma} e : t \qquad G \rhd^{\Sigma} e' : t}{G \rhd^{\Sigma} e \sqsubseteq e' : \mathbf{prop}} \qquad \frac{G \rhd^{\Sigma} h : \mathbf{prop} \qquad G \rhd^{\Sigma} h' : \mathbf{prop}}{G \rhd^{\Sigma} h \Longrightarrow h' : \mathbf{prop}}$$

$$\frac{G, z:\tau \rhd^{\Sigma} h : \tau'}{G \rhd^{\Sigma} \Lambda z{:}\tau.\, h : \tau \to \tau'} \qquad \frac{G, z:\tau \rhd^{\Sigma} h : \mathbf{prop}}{G \rhd^{\Sigma} \epsilon z{:}\tau.\, h : \tau}$$

$$\frac{G \rhd^{\Sigma} h : \tau \qquad G \rhd^{\Sigma} h' : \tau}{G \rhd^{\Sigma} h = h' : \mathbf{prop}}$$

Figure 6.2: LFPC type-checking

6.4 A Logic for FPC

This section extends a higher-order logic with terms from FPC and with constructs from LCF [Paulson, 1987] for expressing properties of the domains of FPC expressions. I call the logic LFPC. Formulae of the logic will be used as sentences in the institution $\mathcal{FPC}$.

6.4.1 Syntax of LFPC

We use a formulation of Higher-Order Logic similar to that described by Gordon and Melham [1993], which takes equality, implication and Hilbert's epsilon as the primitive logical constructs. Other connectives are definable in terms of these. The extension is minimal: FPC types are added as base types of the logic and FPC terms are added to the terms of the logic. The cpo order relation $\sqsubseteq$ is added to express the properties of domains.

Given an FPC signature Σ, the set $\mathit{LogTypes}(\Sigma)$ of types of LFPC, ranged over by τ, is given by the grammar:

$$\tau \ ::= \ t \ \mid \ \mathbf{prop} \ \mid \ \tau \to \tau$$

where $t \in \mathit{ProgTypes}(\Sigma)$. The type equality induced by a signature extends to LFPC types in the obvious way.

Notice that the "logical" function type $s \to t$ is distinct from the function type $s\ {-}\rangle\ t$ in FPC; the latter denotes a cpo of continuous functions, but the former denotes a full set-theoretic function space. Both are needed; abstraction in the programming language can only be over terms of program type, but λ-abstraction is used in the logic to define the connectives and quantifiers, and to construct predicates. Moreover, non-continuous or non-monotonic functions can be useful for writing specifications [Broy et al., 1993b].

Unlike the usual formulation of HOL, there is no infinite type ι of individuals. This is because FPC already has a rich class of types, including types such as $\mathit{nat} =_{\text{def}} \mu a.\mathit{unit} + a$ which denote countably infinite collections. (However, every FPC type has a bottom element; to get rid of it mechanisms of type definition would be useful in the logic, allowing one to define subsets, for example [Gordon and Melham, 1993]. And if the type system were extended with subtyping for such subset definitions, it might alleviate some of the well-known drudgery of LCF-style reasoning with $\bot$.)

Let LogVar be a countable set of variables ranged over by z. We assume $\mathit{LogVar} \supset \mathit{TmVar}$, so that we can abstract over terms of the programming language inside the logic, but not vice-versa. The set of terms of LFPC, ranged over by h, is given by:

$$\begin{array}{lcl} h & ::= & \Lambda z{:}\tau.\,h \mid h(h) \mid z \mid h = h \mid h \Rightarrow h \mid \epsilon z{:}\tau.\,h \\ & \mid & e \mid e \sqsubseteq e \end{array}$$

where e ranges over terms of FPC. Notice that application in the logic is written with parentheses, whereas application in FPC is written by juxtaposition. The usual logical connectives can be defined with the primitives above; definitions are given for reference in Section A.3 on page 271. I will use the derived forms there without further comment.

The set $\mathit{LogTerms}(\Sigma)$ is defined to be the subset of closed terms with types in $\mathit{LogTypes}(\Sigma)$ and term constants in Σ.

Terms of LFPC are type-checked with the rules in Figure 6.2. The rules extend the definition given in Figure 6.1 on page 195, and the type-checking assertion is written in the same way, as $G \rhd^{\Sigma} h : \tau$. Contexts may now include declarations of logical variables, $z : \tau$, which are ignored in the typing and semantics of program terms.

A *formula* is a term with type **prop**. I shall use φ to range over logical terms which are (intended to be) formulae, and Φ to range over sets of formulae.

Remark 6.17. Adding program terms to the logic makes them into "first-class citizens" that can be passed to predicates, etc. But strictly speaking, they don't need to be added: with quantification over program types, which we certainly want for specification, we can express any program term e via the ϵ-operator as $\epsilon x{:}t.\,(x \sqsubseteq e \bigwedge e \sqsubseteq x)$, relying on anti-symmetry for $\sqsubseteq$.

Here is the extension of Proposition 6.4 to typing in LFPC.

Proposition 6.18 (LFPC typing has good properties).

1. *(Closure).*
   If $G \rhd^{\Sigma} h : \tau$, then $FV(h) \subseteq Dom(G)$ and $\tau \in \mathit{LogTypes}(\Sigma)$.
   Moreover, if a term constant v appears in e, then $v \in \Sigma$.

2. *(Type unicity).*
   If $G \rhd^{\Sigma} h : \tau$ and $G \rhd^{\Sigma} h : \tau'$ then $\tau =_{\Sigma} \tau'$.

Proof Standard. □

A signature morphism $\sigma : \Sigma \to \Sigma'$ extends to a translation from $\mathit{LogTerms}(\Sigma)$ to $\mathit{LogTerms}(\Sigma')$ terms, again written as σ. This translation preserves the typing of terms.

Proposition 6.19 (LFPC typing is preserved by signature change).
If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and $G \rhd^{\Sigma} h : \tau$, then $\sigma(G) \rhd^{\Sigma'} \sigma(h) : \sigma(\tau)$.

Proof Easy induction on the derivation of $G \rhd^{\Sigma} h : t$. □

6.5 Semantics of LFPC

The semantics for LFPC is a standard set-theoretic interpretation for higher-order logic (see Gordon and Melham [1993], for example). Each type denotes a non-empty set; the set of truth values is a two element set and an arrow type is interpreted as a set of total functions. Types t of the programming language are interpreted as the underlying set of the cppo $\mathcal{M}[\![t]\!]_{\iota}$.

Definition 6.20 (Interpretation of LFPC types).
Suppose $\tau \in \mathit{LogTypes}(\Sigma)$ and ι is a Σ-type environment. The interpretation of τ is given by structural induction:

- $\mathcal{L}[\![\mathbf{prop}]\!]_{\iota} = \{ \mathit{ff}, \mathit{tt} \}$
- $\mathcal{L}[\![t]\!]_{\iota} = |\mathcal{M}[\![t]\!]_{\iota}|$
- $\mathcal{L}[\![\tau \to \tau']\!]_{\iota} = \mathcal{L}[\![\tau]\!]_{\iota} \to \mathcal{L}[\![\tau']\!]_{\iota}$ □

To interpret logic terms, we need an extended form of term environment which maps variables in LogVar to the domains defined above.

Definition 6.21 (LFPC environments).
Let ι be a Σ-type environment and G an LFPC context. An environment ρ is a (G, Σ, ι)*-LFPC environment* if it maps logic variables $z : \tau$ declared in G to elements of the set $\mathcal{L}[\![\tau]\!]_\iota$, and behaves like a (G, Σ, ι)-FPC environment otherwise.

The interpretation of terms is straightforward. To interpret the choice operator ϵ we assume a choice function which, given any non-empty subset of a type, picks an arbitrary element of that subset. This is used in the final clause of the definition below. Terms from the programming language FPC are interpreted as elements of domains, and $\sqsubseteq$ is interpreted as the approximation relation on the domains.

Definition 6.22 (Interpretation of LFPC terms).
Suppose $G \rhd^\Sigma h : \tau$. Let ι be a Σ-type environment and ρ be a (G, Σ, ι)-LFPC environment. We define the interpretation of h in ι, ρ by induction on the typing derivation:

- $\mathcal{L}[\![G \rhd^\Sigma z : \tau]\!]_{\iota\rho} = \rho(z)$
- $\mathcal{L}[\![G \rhd^\Sigma e : t]\!]_{\iota\rho} = \mathcal{M}[\![G \rhd^\Sigma e : t]\!]_{\iota(\rho|_{TmVar})}$
- $\mathcal{L}[\![G \rhd^\Sigma e_1 \sqsubseteq e_2 : \mathbf{prop}]\!]_{\iota\rho} =$

  $$\begin{cases} tt & \text{if } \mathcal{M}[\![G \rhd^\Sigma e_1 : t]\!]_{\iota\rho} \sqsubseteq_{\mathcal{M}[\![t]\!]_\iota} \mathcal{M}[\![G \rhd^\Sigma e_2 : t]\!]_{\iota\rho} \\ ff & \text{otherwise} \end{cases}$$

- $\mathcal{L}[\![G \rhd^\Sigma h = h' : \mathbf{prop}]\!]_{\iota\rho} =$

  $$\begin{cases} tt & \text{if } \mathcal{L}[\![G \rhd^\Sigma h : \tau]\!]_{\iota\rho} = \mathcal{L}[\![G \rhd^\Sigma h' : \tau]\!]_{\iota\rho} \\ ff & \text{otherwise} \end{cases}$$

- $\mathcal{L}[\![G \rhd^\Sigma h \Longrightarrow h' : \mathbf{prop}]\!]_{\iota\rho} =$

  $$\begin{cases} tt & \text{if } p = tt \text{ and } p' = tt \text{, or if } p = ff \\ ff & \text{otherwise} \end{cases}$$

  where $p = \mathcal{L}[\![G \rhd^\Sigma h : \mathbf{prop}]\!]_{\iota\rho}$ and $p' = \mathcal{L}[\![G \rhd^\Sigma h' : \mathbf{prop}]\!]_{\iota\rho}$

- $\mathcal{L}[\![G \rhd^\Sigma \Lambda z{:}\tau.\, h : \tau \to \tau']\!]_{\iota\rho} =$ the function f where:

  $$f(m) = \mathcal{L}[\![G, z : \tau \rhd^\Sigma h : \tau']\!]_{\iota\rho[z \mapsto m]}$$

  for all $m \in \mathcal{L}[\![\tau]\!]_\iota$

- $\mathcal{L}[\![G \rhd^\Sigma h(h') : \tau']\!]_{\iota\rho} = \mathcal{L}[\![G \rhd^\Sigma h : \tau \to \tau']\!]_{\iota\rho} (\mathcal{L}[\![G \rhd^\Sigma h' : \tau]\!]_{\iota\rho})$

- $\mathcal{L}[\![G \rhd^{\Sigma} \epsilon z{:}\tau.\, h : \tau]\!]_{\iota\rho} =$

$$\begin{cases} \text{any } m \in W & \text{if } W \neq \emptyset, \\ \text{any } m \in \mathcal{L}[\![\tau]\!]_{\iota} & \text{otherwise.} \end{cases}$$

  where

$$W = \left\{ m \in \mathcal{L}[\![\tau]\!]_{\iota} \;\middle|\; \mathcal{L}[\![G, z : \tau \rhd^{\Sigma} h : \mathbf{prop}]\!]_{\iota\rho[z \mapsto m]} = tt \right\} \quad \square$$

The next proposition establishes the satisfaction condition of the institution based on LFPC.

Proposition 6.23 (LFPC meaning is preserved by signature change). *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Suppose ι is a Σ'-type environment and ρ is a $(\sigma(G), \Sigma', \iota)$-LFPC environment. Then*

$$\mathcal{L}[\![G \rhd^{\Sigma} h : \tau]\!]_{\iota|_{\sigma}\rho|_{\sigma}} = \mathcal{L}[\![\sigma(G) \rhd^{\Sigma'} \sigma(h) : \sigma(\tau)]\!]_{\iota\rho}$$

Proof By induction on the definition of $\mathcal{L}[\![-]\!]$, using Proposition 6.16 for the case of FPC terms embedded in LFPC. $\square$

6.6 A Proof System for LFPC

To reason formally about programs in FPC we need a proof system for LFPC. A suitable system consists of axioms and rules for higher-order logic, augmented with an axiomatization of the domain theory used in the fixed-point semantics of FPC expressions. Here we mention some of the axioms for FPC. A putative full proof system is presented for reference in Section A.4 on page 273.

The proof system derives sequents with the form:

$$\Phi \vdash^{\Sigma} \varphi \;\; [G]$$

where Σ is an FPC signature and G is a LFPC context; φ is a Σ-formulae and Φ a set of Σ formulae Φ, each with free variables in G. The context $[G]$ is omitted if it is empty.

First there must be axiom schemes for reflexivity, transitivity, antisymmetry and monotonicity of $\sqsubseteq$, the latter is:

$$\vdash^{\Sigma} \forall f : s \mathrel{-\rangle} t.\, \forall x, y : s.\, x \sqsubseteq y \Rightarrow f\,x \sqsubseteq f\,y$$

Here s and t range over types of FPC, and $s \mathrel{-\rangle} t$ is (or rather, denotes) the continuous function space.

The *existence* of a least fixed point operator is guaranteed by the semantics of the recursive types, and there are FPC terms which correspond to such operators. To use this in the logic, we need to define a canonical fixed point operator. It's also useful to have a canonical bottom element of each FPC type.

$$\begin{array}{lll} \mathit{FIX}_{s,t} & =_{\mathrm{def}} & \mathbf{fun}\,(f : (s \rightarrow t) \rightarrow (s \rightarrow t)). \\ & & \quad (\mathbf{fun}\,(x : r).\ \mathbf{fun}\,(y : s).\, f\ (\mathbf{elim}(x)\ x)\ y) \\ & & \quad (\mathbf{intro}_r(\mathbf{fun}\,(x : r).\ \mathbf{fun}\,(y : s).\, f\ (\mathbf{elim}(x)\ x)\ y)) \\ \multicolumn{3}{l}{\text{where } r \equiv \mu a.a \rightarrow (s \rightarrow t)} \\ \bot_s & =_{\mathrm{def}} & \mathit{FIX}_{s,s}\ (\mathbf{fun}\,(f : s \rightarrow s).\, f) \end{array}$$

Now the properties of *FIX* can be expressed with axioms, for example:

$$\vdash^{\Sigma} \forall f : (s \rightarrow t) \rightarrow (s \rightarrow t).\, \mathit{FIX}_{s,t}\, f \ = \ f(\mathit{FIX}_{s,t}\, f)$$

Further axioms are needed. To summarize, we need to have:

- axioms characterising FPC types as CPOs;
- the rule of fixed point induction;
- axioms for products, sums and function types, including definedness, strictness, distinctness and evaluation properties;
- axioms which characterise the recursive type constructors **intro** and **elim** as witnessing the isomorphism $\mu a.t \cong [\mu a.t/a]t$.

Together, these axioms and rules are enough to derive a large body of theorems about recursively defined datatypes, including principles of structural induction. This exercise has been undertaken in several places [see e.g., Paulson, 1987].

The crucial property of the formal system is of course that it is sound with respect to the semantics. The theorem is stated without proof.

Theorem 6.24 ($\vdash^{\Sigma}$ is sound).
Let ι be a Σ-type environment and ρ be a (G, Σ, ι)-LFPC environment. Suppose $\{\,\varphi_1 \ldots \varphi_n\,\} \vdash^{\Sigma} \varphi\ [G]$. Then

$$\begin{array}{l} \mathcal{L}[\![G \rhd^{\Sigma} \varphi_1 : \mathbf{prop}]\!]_{\iota\rho} = tt \\ \vdots \\ \mathcal{L}[\![G \rhd^{\Sigma} \varphi_n : \mathbf{prop}]\!]_{\iota\rho} = tt \end{array}$$

implies

$$\mathcal{L}[\![G \rhd^{\Sigma} \varphi : \mathbf{prop}]\!]_{\iota\rho} = tt.$$

6.7 The Institution $\mathcal{FPC}$

Now we put together the definitions of previous sections to form the institution $\mathcal{FPC}$. This is straightforward, with the exception that there seems to be no good definition of homomorphism known for algebras with higher-order carriers. Instead we use a discrete model category; anyway, it seems that higher-order logic can capture most uses made of homomorphisms by model-theoretic constructions.

6.7.1 Signatures, $\mathbf{Sign}^{\mathcal{FPC}}$

FPC signatures were defined in Definition 6.1 on page 192 and signature morphisms were defined in Definition 6.5.

We consider *semantic FPC signatures* and *semantic FPC signature morphisms* as equivalence classes of FPC signatures and signature morphisms, with respect to the equalities defined in Definition 6.9. (An equivalent alternative would be to definea semantic signatures as triples $(Ty^{\Sigma}, =_{\Sigma}, Tm^{\Sigma})$).

As a corollary of Proposition 6.6 and Proposition 6.19, equal signatures have the same sets of types, equality relation and sets of program and logic terms. Moreover, if $\Sigma = \Sigma'$, then $Ty^{\Sigma} = Ty^{\Sigma'}$ and $Dom(Tm^{\Sigma}) = Dom(Tm^{\Sigma'})$, and it is easy to show that a signature morphism with domain (or codomain) Σ also has domain (codomain) Σ'. This justifies using equivalence classes of signatures as objects. However, we shall abuse notation by not distinguishing signatures and signature morphisms from the equivalence classes they represent.

Given a signature Σ, the identity signature morphism $id_{\Sigma} : \Sigma \to \Sigma$ is the equivalence class of the inclusion $\iota_{\Sigma,\Sigma}$. Given signature morphisms $\sigma_1 : \Sigma_0 \to \Sigma_1$ and $\sigma_2 : \Sigma_1 \to \Sigma_2$, their composition $\sigma_1 \ ; \ \sigma_2$ is the equivalence class of the componentwise composition $(Ty^{\sigma_1} \ ; \ Ty^{\sigma_2}, Tm^{\sigma_1} \ ; \ Tm^{\sigma_2})$, for some choice of representatives; it is readily shown to be a signature morphism from Σ_0 to Σ_2. Composition is clearly associative.

Thus semantic signatures and signature morphisms form a category, which we call $\mathbf{Sign}^{\mathcal{FPC}}$.

6.7.2 Sentences, $\mathbf{Sen}^{\mathcal{FPC}}$

Terms in the logic LFPC were defined in Section 6.4.1 on page 203. A term φ is a Σ-formula if $\rhd^{\Sigma} \ \varphi : \mathbf{prop}$. The set of Σ-sentences is defined by:

$$\mathbf{Sen}^{\mathcal{FPC}}(\Sigma) =_{\text{def}} \left\{ \varphi \ \middle| \ \rhd^{\Sigma} \ \varphi : \mathbf{prop} \right\}$$

Given a signature morphism $\sigma : \Sigma \to \Sigma'$, we define $\mathbf{Sen}^{\mathcal{FPC}}(\sigma)$ to be the extension of σ to LFPC Σ-terms. By Proposition 6.19, the translation of a Σ-formula is a Σ'-formula.

It is straightforward to show that $\mathbf{Sen}^{\mathcal{FPC}} : \mathbf{Sign}^{\mathcal{FPC}} \to \mathbf{Set}$ is indeed a functor.

6.7.3 *Models,* $\mathbf{Mod}^{\mathcal{FPC}}$

The denotational semantics for FPC terms given in Section 6.3.2 extends to give a notion of algebra for FPC. An FPC algebra interprets type constants as domains from **Dom** and term constants as elements of the appropriate domains.

Definition 6.25 (FPC Algebras).
Let Σ be a signature. A *Σ-algebra A* is a pair (ι_A, ρ_A) where

- ι_A is a Σ-type environment, and
- ρ_A is a (Σ, ι_A)-FPC environment. □

An equivalent view of a Σ-algebra A is as

- a Ty^{Σ}-indexed set $\{\, |A|_c \,\}_{c \in Ty^{\Sigma}}$ of domains in **Dom** such that $c := t \in \Sigma$ implies $|A|_c = \mathcal{M}[\![t]\!]_{\iota}$ for all c, and
- for each $v : t \in \Sigma$, an element $A_v \in \mathcal{M}[\![t]\!]_{\iota}$

where ι is the type-variable environment defined by $\iota(c) = |A|_c$ for all $c \in Ty^{\Sigma}$. We use these two views of algebras interchangeably. Notice that there are no "empty sorts" in this framework.

Definition 6.26 (Reduct of an algebra).
Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism and $A' = (\iota_{A'}, \rho_{A'})$ be a Σ'-algebra. The *σ-reduct* of A' is the Σ-algebra $A'|_{\sigma} =_{\text{def}} (\iota_{A'}|_{\sigma}, \rho_{A'}|_{\sigma})$.

For every signature Σ, the collection of Σ-algebras forms a discrete category denoted $\mathbf{Mod}^{\mathcal{FPC}}(\Sigma)$. For a signature morphism $\sigma : \Sigma \to \Sigma'$, $\mathbf{Mod}^{\mathcal{FPC}}(\sigma)$ is the reduct function $-|_{\sigma}$ of Definition 6.26, which maps algebras in $\mathbf{Mod}^{\mathcal{FPC}}(\Sigma')$ to algebras in $\mathbf{Mod}^{\mathcal{FPC}}(\Sigma)$. It is straightforward to show functorality of $\mathbf{Mod}^{\mathcal{FPC}}$.

6.7.4 *Satisfaction,* $\models^{\mathcal{FPC}}$

We define the satisfaction relation between Σ-algebras $A = (\iota_A, \rho_A)$ and formulae φ by:

$$A \models^{\mathcal{FPC}}_{\Sigma} \varphi \qquad \text{iff} \qquad \mathcal{L}[\![\, \rhd^{\Sigma} \varphi : \mathbf{prop}]\!]_{\iota_A \rho_A} = tt$$

Lemma 6.27 (Satisfaction lemma for $\mathcal{FPC}$).
Let A be a Σ'-algebra and $\sigma : \Sigma \to \Sigma'$ a signature morphism. Then

$$A|_{\sigma} \models^{\mathcal{FPC}}_{\Sigma} \varphi \qquad \textit{iff} \qquad A \models^{\mathcal{FPC}}_{\Sigma'} \sigma(\varphi)$$

Proof Follows directly from Proposition 6.23. □

Combining the previous definitions, we get an institution.

Definition 6.28 (The institution $\mathcal{FPC}$).
The institution $\mathcal{FPC} =_{\text{def}} (\mathbf{Sign}^{\mathcal{FPC}}, \mathbf{Sen}^{\mathcal{FPC}}, \mathbf{Mod}^{\mathcal{FPC}}, \models^{\mathcal{FPC}})$ □

6.8 *Syntax for FPC Signatures and Algebras*

FPC is a bare language of expressions. To construct programs in FPC we need a way of naming and packaging together groups of types and terms. To write specifications, we need a way of packaging axioms together with names of types and terms.

ASL+ provides a language for building specifications using the institution-independent operators of ASL, so all we need at this point is a syntax for writing programs (which denote algebras), a syntax for writing signatures, and a syntax for signature morphisms. Caring about a feasible program and specification language rather than abstract results, we use syntax for finite signatures, and syntax for algebras of finite signature expressible using FPC. (This is in contrast with the ASL syntax defined by Wirsing [1986], for example.) A signature Σ is finite just in case Ty^{Σ} is finite, $Sh^{\Sigma}(c)$ is finite for all $c \in Ty^{\Sigma}$ and $Dom(Tm^{\Sigma})$ is finite.

The main definitions begin in Section 6.8.2. First we consider a motivation behind them: the desire to work in a *context*. This was mentioned in Sections 2.1.5 and 5.8.

6.8.1 Working in a context

Semantically, signatures are always *closed*, in the sense that all type constants referred to are declared in the signature. This is implicit in the definition of FPC signature. However, when writing real programs or specifications, we often work in an *environment* of already-defined types, functions, programs and specifications, perhaps taken from standard libraries. When writing parameterised programs and specifications, or using separate compilation of program parts, we have a *context* of declared programs and specifications; the context corresponds to the formal parameters or module interface. The difference between an environment and a context is that the bindings for the values in an environment are known. Any environment has an underlying context, given by the types which can be inferred for its bindings. Ideally, we should be able to do type-checking with just a context containing typing assumptions; this is in FPC because signatures contain enough information to express sharing.

When writing signatures and algebras, a context is simply an FPC signature. This is because the "core" language FPC knows nothing about the module language; once we move to ASL+ in Chapter 7, a context will also contain components corresponding to specifications, parameterised programs, etc., which have more complex types.

Working in a context, we can relax the closure restriction on syntactic expressions, and use signatures which may not themselves be closed, but are closed when they are added to the context. This is formalized with a signature *extension*, which is a special kind of inclusion in which only new type constants and term constants are added.

Definition 6.29 (Signature extension).
Σ is an *extension* of Σ_{ctx} if both Σ_{ctx} and Σ are signatures, and

- $\Sigma_{ctx} \subseteq \Sigma$
- $Sh^{\Sigma}|_{Ty^{\Sigma_{ctx}}} = Sh^{\Sigma_{ctx}}$
- $Tm^{\Sigma}|_{Dom(Tm^{\Sigma_{ctx}})} = Tm^{\Sigma_{ctx}}$ □

If Σ is an extension of Σ_{ctx}, we think of Σ as a signature-in-context. An algebra-in-context is then given by a function $f : \mathbf{Mod}(\Sigma_{ctx}) \to \mathbf{Mod}(\Sigma)$ which expands any Σ_{ctx}-algebra A to a Σ-algebra $f(A)$, so that $f(A)|_{\Sigma_{ctx}} = A$, and the context part of the algebra is not affected. (Sannella and Tarlecki [1988b] call such a function f a *persistent constructor*).

Because of the syntactic equality = (rather than $=_{\Sigma_{ctx}}$) in the second and third clauses of Definition 6.29, the notion of signature extension is not stable under semantical signature equality. We want this because declaring new terms in a context should not change the context in any way: most importantly, it should not introduce new sharing in the context.[5] Otherwise we could get inconsistencies: there might be an algebra in $\mathbf{Mod}(\Sigma_{ctx})$ which could not be expanded to an algebra in $\mathbf{Mod}(\Sigma)$. Inconsistency ought to be limited to the specification part of the calculus, as far as possible.

Definition 6.30 (Signature morphism in context).
Given two extensions Σ_1, Σ_2 of Σ_{ctx}, a *signature morphism in context* Σ_{ctx} between them is defined to be an FPC signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ such that

$$\begin{tikzcd}
\Sigma_{ctx} \arrow[r, "id"] \arrow[d, "\iota_1"'] & \Sigma_{ctx} \arrow[d, "\iota_2"] \\
\Sigma_1 \arrow[r, "\sigma"'] & \Sigma_2
\end{tikzcd}$$

In words, the action of σ on Σ_{ctx} is the identity.

In the next section, signatures in context are built by the union of a signature for the context Σ_{ctx} with a *pre-signature*. Signature morphisms in context are built using the union of the identity signature morphism $id_{\Sigma_{ctx}}$ with a *pre-signature morphism*, which is a pair of maps defined on the type and term constants declared in a pre-signature.

Notation 6.31.

1. *Let Σ_1 and Σ_2 be pre-signatures. The pre-signature $\Sigma_1 \cup \Sigma_2$ is the "sequential" union of Σ_1 with Σ_2, defined by*

$$\Sigma_1 \cup \Sigma_2 =_{def} (Ty^{\Sigma_1} \cup Ty^{\Sigma_2}, Sh^{\Sigma_1 \cup \Sigma_2}, Tm^{\Sigma_1 \cup \Sigma_2})$$

where

$$Sh^{\Sigma_1 \cup \Sigma_2}(c) = \begin{cases} Sh^{\Sigma_1}(c) & c \in \Sigma_1 \wedge c \notin \Sigma_2 \\ Sh^{\Sigma_2}(c) & c \notin \Sigma_1 \wedge c \in \Sigma_2 \\ Sh^{\Sigma_1}(c) \cup Sh^{\Sigma_2}(c) & c \in \Sigma_1 \wedge c \in \Sigma_2 \end{cases}$$

$$Tm^{\Sigma_1 \cup \Sigma_2}(v) = \begin{cases} Tm^{\Sigma_2}(v) & v \in \Sigma_2 \\ Tm^{\Sigma_1}(v) & v \in \Sigma_1 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

for all $c \in Ty^{\Sigma_1} \cup Ty^{\Sigma_2}$ and $v \in \mathsf{TmConst}$.

[5] It would be possible to relax Definition 6.29 to state just that the sharing in Σ_{ctx} is unchanged, but allowing it to be expressed differently, so giving a definition stable under signature equality. But we would not expect this in the syntax.

2. *We write $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma$ if $\Sigma_{ctx} \cup \Sigma$ is a signature extension of Σ_{ctx}.*

3. *We write $\sigma : [\Sigma_{ctx}]\Sigma_1 \to \Sigma_2$ if $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma_1$, $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma_2$, and*

$$\sigma' =_{def} (Ty^{\sigma} \cup id_{Ty^{\Sigma_{ctx}}}, Tm^{\sigma} \cup id_{Dom(Tm^{\Sigma_{ctx}})})$$

is a signature morphism in context Σ_{ctx} between $\Sigma_{ctx} \cup \Sigma_1$ and $\Sigma_{ctx} \cup \Sigma_2$.

The union operation $\Sigma_1 \cup \Sigma_2$ can be defined in terms of the notation in Notation 6.3 for adding components piecemeal. If Σ_1 and Σ_2 are signatures, then $\Sigma_1 \cup \Sigma_2$ is also a signature, possibly updating points of Tm^{Σ_1} from Tm^{Σ_2}. (Of course, it could be a monster signature having inconsistent type equations, hence an empty class of models.)

6.8.2 Syntax and type-checking

Syntactic signature expressions are finite sequences of declarations of type constants, typed term constants, and sharing equations between type constants and type expressions. Syntactic signature morphisms are sequences of renamings.

Syntactic algebra expressions (programs) are sequences of assignments of types to type names and terms to typed term names $v : t$. The type decoration is added so that every syntactical algebra can be easily associated with a unique syntactical signature. (In some examples the type decoration is omitted, but it should be clear what the intended decoration is.)

The grammar for signatures, signature morphisms, and algebras is:

$$\begin{array}{rcl}
S & ::= & \mathbf{sig}\ sdec^{\star}\ \mathbf{end} \\
sdec & ::= & \mathbf{type}\ c \quad | \quad \mathbf{val}\ v : t \quad | \quad \mathbf{sharing}\ c = t \\
s & ::= & [renam^{\star}] \\
renam & ::= & c \mapsto c \quad | \quad v \mapsto v \\
P & ::= & \mathbf{alg}\ pdec^{\star}\ \mathbf{end} \\
pdec & ::= & \mathbf{type}\ c = t \quad | \quad \mathbf{val}\ v : t = e
\end{array}$$

In examples, sequences are written by juxtaposition.

The rules for type-checking use pre-signatures throughout. A more abstract treatment might use signature extensions rather than pre-signatures; because we have a concrete institution with pre-signatures, we can use "partial" objects rather than morphisms between "full" objects, making the rules easier to read. Pre-signatures are also used in the presentation of ASL+$_{\mathrm{FPC}}$ in Chapter 7.

The judgement forms are:

$$\frac{}{\Sigma_{ctx} \blacktriangleright \; \Rightarrow \emptyset}$$

$$\frac{\Sigma_{ctx} \blacktriangleright sdecs \Rightarrow \Sigma \qquad c \notin (\Sigma_{ctx} \cup \Sigma)}{\Sigma_{ctx} \blacktriangleright sdecs\, \mathbf{type}\; c \Rightarrow \Sigma \cup \{c\}}$$

$$\frac{\Sigma_{ctx} \blacktriangleright sdecs \Rightarrow \Sigma \qquad v \notin (\Sigma_{ctx} \cup \Sigma) \qquad t \in \mathit{ProgTypes}(\Sigma_{ctx} \cup \Sigma)}{\Sigma_{ctx} \blacktriangleright sdecs\, \mathbf{val}\; v : t \Rightarrow \Sigma \cup \{v : t\}}$$

$$\frac{\Sigma_{ctx} \blacktriangleright sdecs \Rightarrow \Sigma \qquad t \in \mathit{ProgTypes}(\Sigma_{ctx} \cup \Sigma) \qquad c \in Ty^{\Sigma}}{\Sigma_{ctx} \blacktriangleright sdecs\, \mathbf{sharing}\; c = t \Rightarrow \Sigma \cup \{c := t\}}$$

$$\frac{\Sigma_{ctx} \blacktriangleright sdecs \Rightarrow \Sigma}{\Sigma_{ctx} \blacktriangleright \mathbf{sig}\; sdecs\; \mathbf{end} \Rightarrow \Sigma}$$

$$\frac{\begin{array}{ll} \Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma & \Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma' \\ \{r_i\} \text{ denotes functions } (Ty^r, Tm^r) & Ty^{\sigma} =_{\text{def}} Ty^r \cup id_{Ty^{\Sigma} - Dom(Ty^r)} \\ Dom(Ty^r) \subseteq Ty^{\Sigma} & Tm^{\sigma} =_{\text{def}} Tm^r \cup id_{Tm^{\Sigma} - Dom(Tm^r)} \\ Dom(Tm^r) \subseteq Tm^{\Sigma} & \sigma : [\Sigma_{ctx}]\Sigma \to \Sigma' \end{array}}{\Sigma_{ctx} \blacktriangleright [r_1 \ldots r_n] \Rightarrow \Sigma \to \Sigma'}$$

Figure 6.3: Type-checking signatures and renamings

$\Sigma_{ctx} \blacktriangleright S \Rightarrow \Sigma$	In Σ_{ctx}, S has pre-signature Σ
$\Sigma_{ctx} \blacktriangleright P \Rightarrow \Sigma$	In Σ_{ctx}, P has pre-signature Σ
$\Sigma_{ctx} \blacktriangleright s \Rightarrow \Sigma \to \Sigma'$	In Σ_{ctx}, s is a renaming from Σ to Σ'

The first two of these are type *inference* judgements, since the pre-signature Σ is determined by the syntactic signature S or the program P. A renaming, on the other hand, does not determine its source or destination signature uniquely.

The rules for type-checking are defined in Figures 6.3 and 6.4. No rebinding of names is allowed in either signatures or algebras.

The rule for adding **sharing** $c = t$ to a signature ensures that c is a type constant declared in the signature, rather than in the context, so the declaration **type** c must appear earlier in the signature. This motivates the abbreviation:

$$\mathbf{type}\; c = t \qquad =_{\text{def}} \qquad \mathbf{type}\; c \quad \mathbf{sharing}\; c = t$$

$$\frac{}{\Sigma_{ctx} \blacktriangleright \ \Rightarrow \ \emptyset}$$

$$\frac{\Sigma_{ctx} \blacktriangleright \ pdecs \ \Rightarrow \ \Sigma \qquad c \notin (\Sigma_{ctx} \cup \Sigma) \qquad t \in ProgTypes(\Sigma_{ctx} \cup \Sigma)}{\Sigma_{ctx} \blacktriangleright \ pdecs \ \ \mathbf{type}\ c = t \ \Rightarrow \ \Sigma \cup \{\, c := t \,\}}$$

$$\frac{\Sigma_{ctx} \blacktriangleright \ pdecs \ \Rightarrow \ \Sigma \qquad v \notin (\Sigma_{ctx} \cup \Sigma) \qquad \begin{array}{c} \rhd^{\Sigma} \ e : t \\ t \in ProgTypes(\Sigma_{ctx} \cup \Sigma) \end{array}}{\Sigma_{ctx} \blacktriangleright \ pdecs \ \ \mathbf{val}\ v : t = e \ \Rightarrow \ \Sigma \cup \{\, v : t \,\}}$$

$$\frac{\Sigma_{ctx} \blacktriangleright \ pdecs \ \Rightarrow \ \Sigma}{\Sigma_{ctx} \blacktriangleright \ \mathbf{alg}\ pdecs\ \mathbf{end} \ \Rightarrow \ \Sigma}$$

Figure 6.4: Type-checking algebras

which was used heavily in the examples in Chapter 2.

In the rule for checking signature morphisms, the renamings must denote a pair of functions from a subset of the type and term constants in the part of the source signature Σ which extends the context Σ_{ctx}. The rest of the mapping is set as the identity on Σ, and the result must be a signature morphism from Σ to the target signature Σ'. In practice this means that a renaming doesn't need to mention every name in a signature; omitted names are mapped to themselves.

The side-conditions in the rules for adding type constant declarations to signatures and algebras ensure that the resulting signature is an extension of the context. This is stated in the proposition below.

Proposition 6.32 (Typing yields signature extensions).
Let Σ_{ctx} be an FPC signature.

1. *If either $\Sigma_{ctx} \blacktriangleright S \Rightarrow \Sigma$ or $\Sigma_{ctx} \blacktriangleright P \Rightarrow \Sigma$ then Σ is unique, and $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma$.*
2. *If $\Sigma_{ctx} \blacktriangleright [r_1 \ldots r_n] \Rightarrow \Sigma \to \Sigma'$ then $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma$ and $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma'$.*

Proof Easy. □

When writing syntax for renamings, it is useful not to have to write syntax also for the source and target signatures. If the source signature is given, then a unique target signature can be constructed as the image of the renaming. This is used in Chapter 7.

Lemma 6.33 (Signature morphism targets).
Given a source signature Σ, if there is a target signature Σ' such that $\Sigma_{ctx} \blacktriangleright s \Rightarrow \Sigma \to \Sigma'$, then there is a unique target signature $s(\Sigma)$, such that $s(\Sigma) \subseteq \Sigma'$ for any other target Σ'.

Proof Define $s(\Sigma)$ as the image of Σ under σ, written $\sigma(\Sigma)$:

$$\sigma(\Sigma) =_{\text{def}} (\sigma(Ty^{\Sigma}), Sh^{\Sigma};\sigma, Tm^{\Sigma};\sigma)$$

where σ is the pair of functions $(Ty^{\sigma}, Tm^{\sigma})$ determined by s as in the rule in Figure 6.3. □

Because signatures were designed to be close to the syntax, we can recover syntactic representations of finite signatures.

Proposition 6.34 (Signatures representations can be recovered).
If $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma$ and Σ is finite, then given orderings on TyConst *and* TmConst*, we can recover a unique syntactic representation $Syntax(\Sigma)$ such that $\Sigma_{ctx} \blacktriangleright Syntax(\Sigma) \Rightarrow \Sigma$.*

Proof The given orderings can be extended to type expressions in some canonical way. Then we can recover a signature expression $Syntax(\Sigma)$ with the form:

```
sig
   type $c_1$   ...
   sharing $c_1 = t_{1,1}$   ...
   val $v_1 : t'_1$   ...
end
```

listing all the types, then the sharing equations, then the value declarations of the signature. □

It won't be used here, but $Syntax(\Sigma)$ would be useful in practical implementations, for reporting inferred types to the user.[6]

6.8.3 *Semantics of signatures, morphisms, and algebras*

Definition 6.35 (Interpretation of syntactic signatures).
The interpretation of a signature expression S in a context Σ_{ctx} is defined to be the signature in context $\Sigma_{ctx} \subseteq_{\mathsf{ext}\cup} \Sigma$ given by the typing judgement $\Sigma_{ctx} \blacktriangleright S \Rightarrow \Sigma$.

[6]A good implementation would attempt to order the declarations as they were input by the user, rather than re-ordering them into a canonical form. For this, FPC signatures should be defined in a less algebraic and more type-theoretic style, even closer to the syntax, i.e., as lists of declarations — see comments in Section 8.2.

Definition 6.36 (Interpretation of syntactic signature morphisms).
The interpretation of a signature morphism expression s in a context Σ_{ctx}, where $\Sigma_{ctx} \blacktriangleright s \Longrightarrow \Sigma \to \Sigma'$, is defined to be the signature morphism in context $\sigma : [\Sigma_{ctx}]\Sigma \to \Sigma'$ determined by s.

Definition 6.37 (Interpretation of syntactic algebras).
The interpretation of an algebra expression P well-typed in a context Σ_{ctx} is defined to be the functor $f_P : \mathbf{Mod}(\Sigma_{ctx}) \to \mathbf{Mod}(\Sigma_{ctx} \cup \Sigma)$ given by

$$f_P(A) = \mathcal{P}[\![\Sigma_{ctx} \blacktriangleright P \Longrightarrow \Sigma]\!]_{(\iota_A, \rho_A)}$$

where the interpretation $\mathcal{P}[\![\Sigma_{ctx} \blacktriangleright P \Longrightarrow \Sigma]\!]_{(\iota_A, \rho_A)}$ is defined using induction on the typing derivation of $\Sigma_{ctx} \blacktriangleright \mathit{pdecs} \Longrightarrow \Sigma$, with $P \equiv$ **alg** *pdecs* **end**:

- $\mathcal{P}[\![\Sigma_{ctx} \blacktriangleright \;\Longrightarrow \varnothing]\!]_A = A$
- $\mathcal{P}[\![\Sigma_{ctx} \blacktriangleright \mathit{pdecs}\;\; \mathbf{type}\, c = t \Longrightarrow \Sigma \cup \{\, c := t \,\}]\!]_{(\iota_A, \rho_A)}$
  $= (\iota_A[c \mapsto \mathcal{M}[\![t]\!]_{\iota_A}],\ \rho_A)$
- $\mathcal{P}[\![\Sigma_{ctx} \blacktriangleright \mathit{pdecs}\;\; \mathbf{val}\, v : t = e \Longrightarrow \Sigma \cup \{\, v : t \,\}]\!]_{(\iota_A, \rho_A)}$
  $= (\iota_A,\ \rho_A[v \mapsto \mathcal{M}[\![\, \rhd^{\Sigma}\ e : t]\!]_{\iota_A, \rho_A}])$ □

6.9 Discussion and Related Work

This section begins with a comparison to the related work, in Sections 6.9.1 and 6.9.2, and then discusses some points of the design of the institution $\mathcal{FPC}$ and its syntax, in Section 6.9.3.

There is not much, if anything, in the literature which aims for the same breadth as the work begun here: putting together a programming language, specification language, program logic, and higher-order module system. However there are comparisons between the institution $\mathcal{FPC}$ and some other work. The next two subsections discuss some related logics and specification languages.

6.9.1 Related logics

The idea of embedding an LCF-style logic in a higher-order logic has occurred to several researchers over the last few years. Here I want to mention three other projects. The first two, HOL-CPO and HOLCF, are implementations of logics; the third, $\lambda\omega_L^{\mu}$, is more theoretical and also considers a programming language. None of them is oriented towards design of a specification language.

HOL-CPO Sten Agerholm [1994a, 1994b] has implemented the logic of LCF in the HOL theorem prover [Gordon and Melham, 1993], by definitional extension. This has the benefits of being able to formalise admissibility within the logic (see comments in Section 6.9.3). A cpo is formalized as a pair of a predicate, describing a subset of a HOL type, and an order relation on the type.

HOLCF Regensburger [1994] has implemented another axiomatisation of LCF by extension of Isabelle/HOL, the implementation of the HOL logic inside the Isabelle theorem prover [Paulson, 1994]. His axiomatisation is more sophisticated than Agerholm's, making use of Isabelle's order-sorted polymorphism to define a type class for cpos, so a cpo is simply a type of the logic equipped with some special operations and properties.

$\lambda\omega_L^\mu$ Poll [1994] describes the hand-in-hand construction of a family of programming languages and corresponding programming logics, each with higher-order polymorphism. He uses the concise framework of Pure Type Systems [Barendregt, 1992] to describe the systems, formulating the dependencies to prevent logical operations appearing in programs. The final system is $\lambda\omega_L^\mu$, which is a programming language with recursive types and a fixed-point operator tied together with an LCF-like extension to a higher-order logic.

The rule-structure of the PTS defining $\lambda\omega_L^\mu$ allows program terms as subterms in the logic, but terms of the logic (of program type) cannot occur in program terms. For us, LFPC makes the distinction by making the program types a subset of the logical types. Poll's approach sacrifices flexibility: he is cannot write a function mapping a chain (encoded as a predicate) to its LUB, for example, because the type of such a function is disallowed in the framework [Poll, 1994, p.95]. LFPC does not suffer from this drawback.

A major difference to Poll's treatment is that while LFPC is a logic for FPC with a call-by-value operational semantics, $\lambda\omega_L^\mu$ must allow an arbitrary beta-reduction order in its programming language, because the same type-theoretic framework, which does not distinguish reduction order, is used for writing both programs and specifications.[7]

[7]This criticism applies equally to other type-theoretic approaches which model program computation using the reduction of the type theory.

6.9.2 Related specification languages

The specification language based on $\mathcal{FPC}$ appears as part of ASL+$_{\mathrm{FPC}}$ in the next chapter, but we are almost there: the parts of the syntax defined in Section 6.8 provide the institution-dependent syntax needed for ASL expressions — see Section 1.2. Here is a quick mention of some related specification languages.

EML [Kahrs et al., 1994] was mentioned in Section 1.4. The module parameterisation of ASL+ was invented to capture and generalise that of EML. The language FPC can be seen as a restricted fragment of the core-level language of SML, containing the power of recursive functions and non-parameterised datatype declarations, but without polymorphism, exceptions, or assignment. Attempts towards giving EML an algebraic semantics are unfinished as yet [Sannella and Tarlecki, 1986, 1989].

SPECTRUM Manfred Broy's group in Munich has developed a specification language with an institutional semantics, having higher-order functions and order-sorted polymorphism [Broy et al., 1993a,b]. The logic of SPECTRUM is 3-valued. It does not have a module language, but the order-sorted polymorphism compensates to some extent [Grosu and Nazareth, 1994].

Metasoft Tarlecki [1992] describes a module system for the Metasoft specification language which includes the idea of having "symbolic type expressions" in signatures, also one of the innovations in $\mathcal{FPC}$.

This idea was re-invented lately, in studies of type systems for the SML module system which add "manifest types" to signatures [Leroy, 1994, Harper and Lillibridge, 1994]. These type systems are discussed more in the next chapter.

Hofmann and Sannella Hofmann and Sannella [1996] define the basic components of an institution closely related to $\mathcal{FPC}$. They do not provide a syntax for signatures and algebras, however; their motivation is to study behavioural equivalence in a higher-order setting.

6.9.3 Remarks on language design

Here are a few remarks about the design of $\mathcal{FPC}$, continuing some comparisons to the related work.

Stratification of levels

Syntactical signatures and algebras are similar to record types and values. Why not simply add these to FPC? There are two reasons.

First, algebra expressions allow bindings of identifiers to types as well as to values, so they are more than ordinary record types. This could be accommodated by a richer notion, such as sigma-types (dependent products), existential types, or dependent records [Luo, 1993, Mitchell and Plotkin, 1988, Nordström et al., 1990]. But the underlying type-system would have to be changed; we would no longer be in the world of simple types and we would have to consider a more complicated program semantics.

The second reason concerns a tacit aspect of ASL+: it imposes a *stratification* of design levels in programming and specification. The programming language is not expected to incorporate modules as values which can be computed with. Modules appear at a higher level, so the modular structure of a program is essentially something static that doesn't change during the course of its execution. (This can be contrasted with object-oriented programs, where objects are in some sense modules, and may be created and destroyed dynamically while a program executes.) Of course, the modular structure of a specification is also static.

From an engineering point of view, this stratification of levels may be desirable. The strategy is exemplified for programming languages by SML and explained by MacQueen [1991]. Using richer type-theoretic constructs for modules would lose the stratification.

Having said it may be desirable, the separation of levels here is somewhat extreme: it causes difficulty in Chapter 7, because the core-level language has no reference to the module-level context.

Embedding syntax or semantics

Both of the logics HOL-CPO and HOLCF mentioned in Section 6.9.1 are formalizations of the *semantics* of LCF. In these theories, the term $FIX(f)$ is *defined* to be $\bigsqcup f^n(\bot)$. Classical rules of LCF like fixed point induction can then be derived.[8]

In contrast, the approach in this chapter is an direct embedding of the syntax of FPC inside the logic, adding axioms to HOL which reflect the LFPC

[8]For recursive types, neither HOL-CPO and HOLCF have an axiomatization, and the solution of recursive domain equations within the logic itself is not possible. As in LCF, the axiomatisation must be extended when recursively-defined types are added (typically, tools are written to help with this).

semantics, just as LCF adds axioms to first order logic. However, the axiomatization here goes beyond LCF, for example because we can express the semantic property of admissibility. This is claimed to be one of the major advantages of HOL-CPO and HOLCF. In the first-order world of LCF, admissibility had to be hard-coded as an incomplete set of syntactic tests and heuristics [Paulson, 1987].

Whilst an embedding of the semantics is theoretically heart-warming, it seems more feasible for practical development tools to assert axioms directly from real programs, rather than using a translation to reconstruct logically equivalent models and terms inside the theorem prover. Embedding only the syntax reflects this more direct approach, where we reason directly with the syntax of programs. And because the syntax of FPC is not an object of discourse in the logic, this doesn't result in contraditions: it is safe to assert that $pred(succ(0)) = 0$, for example.

Internal and external program logics

The logic LFPC is used to reason about programs written in FPC, which is a separate language (although arbitrary FPC expressions are allowed in LFPC). This is an example of the so-called *external* approach, like Erik Poll's system $\lambda\omega_L^\mu$ mentioned in Section 6.9.1. In fact, $\mathcal{FPC}$ is more separated than $\lambda\omega_L^\mu$, which has the advantages mentioned before.

Some researchers argue that for program development, an *internal* approach is preferable, where a single language encompasses both programs and specifications. In type theory, for example, programs arise as a special case of proofs. The advantage is that a program and its correctness proof are synthesized together, so there is no duplication of work (writing a loop, then using an inductive proof, etc.). The disadvantage is that the structure of programs and proofs is *forced* to be the same, and there is a need to extract programs from proofs somehow (and then a need to optimise them, since the simplest proofs rarely correspond with the most efficient programs).

Erik Poll discusses more about the internal and external approaches in his thesis [Poll, 1994]. It's worth saying that although $\mathcal{FPC}$ uses the external approach, nothing in ASL+ forces this: one could instead begin from an institution in which sentences are propositions in a constructive type theory.

6.10 Summary

This chapter described the construction of an institution $\mathcal{FPC}$ for ASL+, based on FPC as a programming language, described in Section 6.2, and HOL extended with LCF as a program logic, described in Section 6.4.

When I began this project, I imagined that I would easily find a definition in the literature for an institution similar to $\mathcal{FPC}$. It was surprising to discover that similar attempts are few and far between; most work in algebraic specification is restricted to first-order languages, for which the semantic basis is better understood. So the work on $\mathcal{FPC}$ was started as an exercise in pushing the definitions through to examine how the institution-independent framework deals with a concrete instance of ASL+, based on a realistic setting — something like a subset of Standard ML. Perhaps this is the first time that this exercise has been carried out for a language like FPC.

The construction of $\mathcal{FPC}$ has been fully formalized, finishing with a syntax for writing signatures, signature morphisms, and algebras in Section 6.8. These parts of the syntax are not required by the definition of institution, but they are required to build a syntax for a programming language and a specification language based on one. (In particular, we could get a complete syntax for ASL expressions in $\mathcal{FPC}$— see Section 1.2.)

An unusual feature was the use of type equations in the definition of FPC signature, Definition 6.1. As outlined in Chapter 5, this allows sharing to be accommodated directly in an institutional semantics, without needing to introduce special mappings from program-level identifiers to semantical names. The set of sentences $\mathbf{Sen}^{\mathcal{FPC}}(\Sigma)$ consists exactly of the well-typed propositions in Σ.

One of the lessons of this exercise is the importance of type-checking, as a necessary step from semantic foundations towards practical applications using concrete syntax. Type-checking rules were given for each core-language part of the syntax picture shown in the introduction. The lesson will be reinforced, which completes the picture by studying type-checking in the module language ASL+_{FPC}.

7 ASL+ based on $\mathcal{FPC}$

This chapter describes an instance of ASL+ called ASL+$_{\text{FPC}}$, based on the institution $\mathcal{FPC}$ defined in Chapter 6. The language ASL+$_{\text{FPC}}$ tackles the context and sharing problems described in Chapter 5. Using some of the concrete details of $\mathcal{FPC}$, the result is a language with a better notion of context and a more powerful rough type system which can express type propagation across module boundaries. The examples in Chapter 2 work directly in ASL+$_{\text{FPC}}$.

This is an experimental first-step towards an expressive type system, rather than a robust and final solution.

7.1 Modules for FPC

WITH A PARTICULAR INSTITUTION to hand, we can learn from the specifics before generalising to an abstract setting. In this chapter I shall design a version of ASL+ based on the formulation of $\mathcal{FPC}$ in Chapter 6. It is dubbed ASL+$_{\text{FPC}}$.

The language ASL+$_{\text{FPC}}$ tackles the problems which the abstract version of ASL+ in Chapter 5 was criticised for. The new language has a built-in notion of context, given by a dot-notation renaming mechanism. It can explain the propagation of type identity in the rough type-checking system, because rough types may contain identifiers (inside signatures), and the arrow type is changed to a dependent product. The examples given in Chapter 2 work directly in ASL+$_{\text{FPC}}$.

In fact only a few places in this chapter do rely on the specifics of $\mathcal{FPC}$, but studying the details in a concrete setting has helped to understand the problems. Most of the time spent here has been syntactic skullduggery: designing a language in which the ASL+ constructs are transformed from heavily-typed semantical constructions into lightweight syntax, which works for real examples. Some of the particular problems involved, and several of the devices used to solve them, are original, because ASL+ has new parameterisation constructions not present in other languages. But I also draw on recent research into type systems for programming language modules [including: Leroy, 1994, 1995, 1996a,b, Courant, 1997a] and learnt much from discussions with a colleague at Edinburgh who is writing a thesis on a related topic [Russo, 1997].

The type system given here is not the end of the story, by any means; it has some bad features and some features missing, which are discussed in the final sections. I think it is an experimental step in the right direction.

The chapter begins with the syntax and rough type-checking systems for ASL+$_{\mathrm{FPC}}$, in Section 7.2. The definitions given here may appear ad hoc, so I have included a commentary on how the design was reached, as an interlude in Section 7.3; it may help to refer to it while reading Section 7.2. The commentary also includes some mention of related module systems. Returning to the definitions, roughly-typable terms are given a set-theoretic semantics in Section 7.4, and some early ideas for a type-system for proving satisfaction are given in Section 7.5. Some of the difficulties with the system are revealed in Section 7.6, which also mentions some related work. Finally, Section 7.7 summarizes.

7.2 Syntax

The syntax and rough type-checking for ASL+$_{\mathrm{FPC}}$ begins in Section 7.2.1. Before then, I give some auxiliary definitions. Throughout this chapter, Σ ranges over *pre*-signatures of FPC, and all talk of (semantic) "signatures" will refer pre-signatures unless stated. Pre-signatures were already used in the type-checking rules for algebras and signatures in Section 6.8.2.

Operations on signatures

The syntax uses a new set of variables *ModVar*, the *module variables*, ranged over by $X, Y, Z, \ldots$. We treat the dot operator as a special "prefix" name constructor operator on the type and term constants of FPC:

$$
\begin{array}{lll}
\_.\_ & : & \mathit{ModVar} \times \mathit{TyConst} \to \mathit{TyConst} \\
\_.\_ & : & \mathit{ModVar} \times \mathit{TmConst} \to \mathit{TmConst}
\end{array}
$$

such that $X.v \not\equiv v$ and whenever $X.v \equiv Y.v'$ then $X \equiv Y$ and $v \equiv v'$, for all $X, Y \in \mathit{ModVar}$ and for all type or term constants v, v'. In other words, the prefix of a constant by an algebra variable yields a new constant, and two syntactically equal "dotted constants" must have been constructed from equal components. The idea is that in an implementation, the dot is a special character which is not normally allowed in names of identifiers.

Definition 7.1 (Dot renamings with respect to a signature).

1. The dotting operation extends to types and terms over a signature. We define

$$
\begin{array}{lll}
X_\Sigma.t & =_{\mathrm{def}} & t[X.c, \ldots / c, \ldots] \\
X_\Sigma.e & =_{\mathrm{def}} & e[X.v, \ldots / v, \ldots] \\
X_\Sigma.h & =_{\mathrm{def}} & h[X.v, \ldots / v, \ldots]
\end{array}
$$

   where the above notation indicates the simultaneous prefixing of all constants $c, v \in \Sigma$ (v ranges over both type and term constants). Constants not in Σ are not affected.

2. The dotting operation extends to give a renaming operation on signatures by prefixing all the names:

$$
\begin{array}{ll}
X.\Sigma =_{\mathrm{def}} & (\{ X.c \mid c \in \Sigma \}, \\
& \{ c \mapsto \{ X_\Sigma.t \mid c := t \in \Sigma \} \mid c \in \Sigma \}, \\
& \{ X.v \mapsto X_\Sigma.t \mid v : t \in \Sigma \})
\end{array}
$$

□

It is easy to see that the extension of the dotting operation to signatures is an isomorphic renaming, such that the types and terms over the renamed signature are disjoint with those over the original. Definition 7.1 also makes sense for pre-signatures.

As well as the full-blown dot renaming of a signature, we require a slightly different operation which adds extra sharing equations to a signature.

Definition 7.2 (Strengthening of a signature).
Given a pre-signature Σ and a variable $X \in \mathit{ModVar}$, we define the *strengthening of Σ by X*, written Σ / X, to be the signature $(\mathit{Ty}^\Sigma, \mathit{Sh}, \mathit{Tm}^\Sigma)$ where the new sharing component Sh is defined for $c \in \mathit{Ty}^\Sigma$ by:

$$\mathit{Sh}(c) = \mathit{Sh}^\Sigma(c) \cup \{ X.c \}.$$

This operation is used when projecting a variable from the context. When $X : \Sigma$ appears in the context, we can use X to denote a Σ-algebra, knowing that for every type c in X, the equation $c = X.c$ holds.

Definition 7.3 (Sharing-only subsignature).
We write $\Sigma_1 \subseteq_{\mathsf{sh}} \Sigma_2$ if

- $\Sigma_1 \subseteq \Sigma_2$
- $Ty^{\Sigma_1} = Ty^{\Sigma_2}$
- $Tm^{\Sigma_1} = Tm^{\Sigma_2}$

If $\Sigma_1 \subseteq_{\mathsf{sh}} \Sigma_2$, then Σ_2 only differs from Σ_1 in having more sharing.

Substitution and reduction

The use of substitution in the rough type-checking system is restricted to renaming one variable for another. Constants $X.v$ are renamed to constants $Y.v$ throughout an expression, for example.

Applications are only allowed to have variables as operands, so the only kind of β-redex which appears is:

$$(\lambda X{:}A.\, M)\; Y \longrightarrow_\beta M[Y/X]$$

and the reduct is simply the renaming of X by Y in M. Reduction involves replacement in algebra expressions, signature expressions, and formulae; as usual, we may need to rename to avoid clashes. We omit the full definition, which would be given by induction over the structure of terms shown in the next section.

7.2.1 Syntax and rough type-checking

The pre-contexts and pre-terms of ASL+$_{\mathrm{FPC}}$ are formed from the grammar:

$$\begin{array}{rcl}
\Gamma & ::= & \langle\rangle \;\mid\; \Gamma, SP \;\mid\; \Gamma, X : A \;\mid\; \Gamma, X = M \\
M, SP, A, B & ::= & X \;\mid\; P \;\mid\; S \\
& \mid & \textbf{impose } \varphi \textbf{ on } SP \\
& \mid & \textbf{derive from } SP \textbf{ by } s : S \\
& \mid & \textbf{translate } SP \textbf{ by } s \\
& \mid & \textbf{enrich } SP \textbf{ with } SP \\
& \mid & \lambda X{:}A.\, M \;\mid\; M\, X \\
& \mid & \Pi X{:}A.\, B \;\mid\; \mathsf{Spec}(A) \\
& \mid & [X = M]M : A
\end{array}$$

Variables $X, Y, Z, \ldots$ are taken from the set *ModVar*. Recall that S ranges over signature expressions of FPC, P over algebra expressions, and s over signature morphism expressions. These were defined in Section 6.8. The variable φ ranges over terms of LFPC, as defined in Section 6.4.1. Since the logic LFPC has conjunction and the truth value T, a finite set of formulae Φ can be replaced with a single formula φ in **impose**.[1]

The meta-variables SP, A, B, M, N and variants will all be used to range over the set of pre-terms. The intuition is that SP stands for terms which turn out to denote "base" specifications (collections of FPC algebras), whilst A, B stand for arbitrary collections and M, N for arbitrary terms.

As well as allowing declaration and bindings of variables, contexts can be directly extended by base specification expressions. This is to allow the specification of "pervasive" datatypes of the language which are visible everywhere, such as BOOLEAN. As usual, a pre-context cannot contain repeated declarations of the same variable.

In contrast with the abstract formulation of ASL+ in Chapter 5, I have included a specific subset of the specification building operators of ASL, as well as the context-sensitive syntax for signatures and algebras from FPC. We use a primitive **enrich** operator instead of considering **enrich** to be defined via **translate** and union. Unions, along with the other ASL operators, are left for future treatment. (Alternatively, it might be better to consider a small group of higher-level constructs — see discussion in Section 7.6.1 on page 258.)

The pre-terms of the λ-calculus fragment are as in ASL+, except that application $M\,X$ is only to variables, and singleton types are omitted. The pre-term $[X = M]N : A$ is a local binding construct; X is bound (to M) in N, but is not bound in M or A.

Rough types and rough contexts

The syntax is:

$$\begin{array}{lcl} \tau & ::= & \Sigma \quad | \quad \pi X : A.\tau \quad | \quad P(\tau) \\ \Gamma & ::= & \langle\rangle \quad | \quad \Gamma, \Sigma \quad | \quad \Gamma, X : \tau \end{array}$$

where Σ ranges over pre-signatures. As usual, we may write $A \to \tau$ for $\Pi X{:}A.\,\tau$ when X does not appear bound in τ.

Rough types which differ only in the choice of bound variables are considered identical. Substitution on terms was explained at the start of this

[1]This is a minor simplification to avoid cluttering the presentation; a real specification language would prefer to keep axioms separate, and also to name them.

section; substitution and renaming for rough types is defined similarly. Notice that this involves substitution or renaming inside semantic signatures Σ, and that the notion of bound and free module variables is extended to consider the variables appearing in signatures.

Unlike the rough typing systems considered before which used "full" contexts, we now have a separate syntax for rough contexts. This is simply for convenience because rough typing is more complicated than before. I use Γ to range over both rough contexts and "full" contexts, but it should be clear which is intended; when they are used close by, Γ_R is used for the rough context.

Rough typing judgements

There are five judgement forms:

$\Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma$	Σ_Γ is the underlying FPC signature of Γ
$\Gamma \blacktriangleright \tau$	τ is a well-formed rough type in Γ
$\blacktriangleright \Gamma$	Γ is a rough typing context
$\Gamma \blacktriangleright \tau \leq \tau'$	τ is a subtype of τ'
$\Gamma \blacktriangleright M \Rightarrow \tau$	M has rough type τ

These judgements are defined in Figures 7.1–7.4. In all cases, Γ is a rough context. The rules in each figure are described below.

Formation rules (Figure 7.1).
The first four rules in this table construct the underlying FPC signature of a context. This is done by combining the pre-signatures for the pervasive parts of the context, together with the dot-renamed components $X.\Sigma$ for variables X which range over Σ-algebras.[2] Module variables which have non-signature types do not contribute to the FPC signature of the context, since there is no way to use them directly in any FPC type or term.

The three rules for forming rough types are straightforward. In the case of a pre-signature, we must check that it is an extension of the signature of the context. Formation of contexts is standard.

Subtyping rules (Figure 7.2).
The subtyping rules lift the sharing-only subsignature relation to a relation on rough types. The rule for Π-rough types is the usual contravariant rule; for power rough types, the relation is covariant. Semantically this captures an inclusion on the domains interpreting rough types.

[2]This is a sort of "flattening" operation, since FPC does not have "nested algebras" akin to nested modules in programming languages.

Programs and ASL terms (Figure 7.3).
The rules for rough typing ASL terms, including FPC signatures and algebras, involve some signature calculation. The first two rules, introducing signatures and programs into the system, are straightforward; they invoke the type-checking system for the core-level from Section 6.8. The rule for **impose** is also straightforward; it invokes the type-checking rules for LFPC in Section 6.4.1 to be sure that φ is a well-typed proposition.

The rules for **derive** and **translate** use syntax for signature morphisms, which allows some polymorphism. The argument of **derive from** – **by** $s : S$ or of **translate** – **by** s can have any signature which fits suitably with s, according to the type-checking rules in Figure 6.3 on page 215. The result signature of **derive** has to be given, but the result of **translate** is inferred, as the smallest image $s(\Sigma)$ of s, defined in Lemma 6.33 on page 217.[3] The ideas behind these rules are explained more in Section 7.3.6.

The rule for **enrich** is similar to a rule for the dependent sum in type theory: just as x occurs bound in B in the term $\Sigma x{:}A.B$, so all the symbols of SP occur bound in SP' in the term **enrich** SP **with** SP'. The non-symmetry in **enrich** isn't shown by its usual definition in terms of **translate** and **union** (see page 12), but here the directly-defined semantics of **enrich** SP **with** SP' shows the dependency: models of the result are extensions of models of SP.

ASL+ terms (Figure 7.4).
The rough type of a variable X which has signature type Σ is the strengthened version of Σ which reflects the sharing of X with the context, since it denotes a projection on the X.-named part of the underlying environment.

Functions can only be applied to variables. To use the application rule it may be necessary to rename the bound variable of the Π-type of the function to match the operand.[4] The application rule is the crucial places for allowing propagation of type identities: the use of subtyping here allows the actual parameter to have a richer type with more sharing equations than the type of the formal parameter A. Propagation of the type identities occurs because after application any mention of $X.c$ in the result type τ' will refer to a variable declared in the context, possibly having more sharing equations, rather than the bound variable of the Π-type.

The rule for a binding $[X = M]N : A$ allows N to be typed in the rough context extended by the typing of M, but the rough type of A has to be

[3]This means that **translate** can only be used with surjective signature morphisms. But we can express translation along inclusions **translate** SP **by** $\iota : \Sigma \hookrightarrow \Sigma'$ using an enrichment **enrich** SP **with** S, where $S = \mathit{Syntax}(\Sigma' - \Sigma)$.

[4]A couple of people have mentioned that this renaming idea seems like a cunning and potentially useful trick; probably it has been used elsewhere in λ-calculi implementations, but I don't know where.

valid in the context Γ, so the dependency on X must be removed. The type in the extended context should be a subtype of the type of A. This rule is restricted to algebras or specifications in the body N, but the restriction could perhaps be lifted at the cost of extra complication (to define projection and weakening operations on environments).

The other rules in Figure 7.4 are much the same as rough typing rules seen for λ_{Power} in Section 4.5, except that λ- and Π-terms now have dependent rough types.

7.2.2 Expected properties of rough typing

Detailed proofs for this section have not yet been completed, so the statements below should be regarded as **conjectures** at this time.

Proposition 7.4 (Formation).

1. *Suppose one of $\Gamma \blacktriangleright \tau$, $\Gamma \blacktriangleright \tau \leq \tau'$, or $\Gamma \blacktriangleright M \Longrightarrow \tau$. Then there is a subderivation of $\blacktriangleright \Gamma$.*
2. *If $\blacktriangleright \Gamma$, then for some Σ_Γ, $\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma$ and $\Sigma_\Gamma \in \mathbf{Sign}^{\mathcal{FPC}}$.*
3. *If $\Gamma \blacktriangleright M \Longrightarrow \tau$, $\Gamma \blacktriangleright \tau \leq \tau'$ or $\Gamma \blacktriangleright \tau' \leq \tau$ then $\Gamma \blacktriangleright \tau$.*

Proof Induction on derivations. □

Proposition 7.5 (Bound narrowing).
Let $\Gamma_1 \equiv \Gamma, X : \tau_1, \Gamma'$ and $\Gamma_2 \equiv \Gamma, X : \tau_2, \Gamma'$ with $\Gamma \blacktriangleright \tau_2 \leq \tau_1$. Then

1. *If $\Gamma_1 \Longrightarrow_{\mathsf{sig}} \Sigma_{\Gamma_1}$, then there is Σ_{Γ_2} such that $\Gamma_2 \Longrightarrow_{\mathsf{sig}} \Sigma_{\Gamma_2}$ and $\Sigma_{\Gamma_1} \subseteq_{\mathsf{sh}} \Sigma_{\Gamma_2}$.*
2. *If $\Gamma_1 \blacktriangleright J$ then $\Gamma_2 \blacktriangleright J$, where J represents a (type or context) formation or subtyping judgement.*
3. *If $\Gamma_1 \blacktriangleright M \Longrightarrow \upsilon_1$ then there is υ_2 such that $\Gamma_2 \blacktriangleright M \Longrightarrow \upsilon_2$ and $\Gamma_2 \blacktriangleright \upsilon_2 \leq \upsilon_1$.*

Proof By induction on derivations, using properties of the union operator on signatures. □

Theorem 7.6 (Closure under reduction).
If $\Gamma \blacktriangleright M_1 \Longrightarrow \tau_1$ and $M_1 \twoheadrightarrow_\beta M_2$, then $\Gamma \blacktriangleright M_2 \Longrightarrow \tau_2$ for some τ_2 such that $\Gamma \blacktriangleright \tau_2 \leq \tau_1$.

Proof Using Proposition 7.5. □

$$\frac{}{\langle\rangle \Rightarrow_{\mathsf{sig}} \varnothing}$$

$$\frac{\Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma}{\Gamma, \Sigma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \cup \Sigma}$$

$$\frac{\Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma}{\Gamma, X : \Sigma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \cup X.\Sigma}$$

$$\frac{\Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \tau \text{ is not a signature}}{\Gamma, X : \tau \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma}$$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \subseteq_{\mathsf{ext}\cup} \Sigma}{\Gamma \blacktriangleright \Sigma}$$

$$\frac{\Gamma \blacktriangleright A \Rightarrow \mathsf{P}(\tau) \qquad \Gamma, X : \tau \blacktriangleright \tau'}{\Gamma \blacktriangleright \pi X : A.\tau'}$$

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

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

$$\frac{\Gamma \blacktriangleright \Sigma}{\blacktriangleright \Gamma, \Sigma}$$

$$\frac{\Gamma \blacktriangleright \tau}{\blacktriangleright \Gamma, X : \tau}$$

Figure 7.1: Formation rules of ASL+$_{\mathrm{FPC}}$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \begin{array}{c}\Sigma_\Gamma \subseteq_{\mathsf{ext}\cup} \Sigma, \quad \Sigma_\Gamma \subseteq_{\mathsf{ext}\cup} \Sigma' \\ (\Sigma_\Gamma \cup \Sigma') \subseteq_{\mathsf{sh}} (\Sigma_\Gamma \cup \Sigma)\end{array}}{\Gamma \blacktriangleright \Sigma \le \Sigma'}$$

$$\frac{\begin{array}{ll}\Gamma \blacktriangleright A_1 \Rightarrow \tau_1 & \Gamma \blacktriangleright \tau_2 \le \tau_1 \\ \Gamma \blacktriangleright A_2 \Rightarrow \tau_2 & \Gamma, X:\tau_2 \blacktriangleright \tau_1' \le \tau_2'\end{array}}{\Gamma \blacktriangleright \pi X:A_1.\tau_1' \le \pi X:A_2.\tau_2'}$$

$$\frac{\Gamma \blacktriangleright \tau \le \tau'}{\Gamma \blacktriangleright P(\tau) \le P(\tau')}$$

Figure 7.2: Subtyping rules for rough types of ASL+$_{\mathrm{FPC}}$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \blacktriangleright S \Rightarrow \Sigma}{\Gamma \blacktriangleright S \Rightarrow P(\Sigma)}$$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \blacktriangleright P \Rightarrow \Sigma}{\Gamma \blacktriangleright P \Rightarrow \Sigma}$$

$$\frac{\Gamma \blacktriangleright SP \Rightarrow P(\Sigma) \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \rhd^{\Sigma_\Gamma \cup \Sigma} \varphi : \mathbf{prop}}{\Gamma \blacktriangleright \textbf{impose } \varphi \textbf{ on } SP \Rightarrow P(\Sigma)}$$

$$\frac{\Gamma \blacktriangleright SP \Rightarrow P(\Sigma') \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \begin{array}{l}\Sigma_{ctx} \blacktriangleright S \Rightarrow \Sigma \\ \Sigma_\Gamma \blacktriangleright s \Rightarrow \Sigma \to \Sigma'\end{array}}{\Gamma \blacktriangleright \textbf{derive from } SP \textbf{ by } s:S \Rightarrow P(\Sigma)}$$

$$\frac{\Gamma \blacktriangleright SP \Rightarrow P(\Sigma) \qquad \Gamma \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \blacktriangleright s \Rightarrow \Sigma \to s(\Sigma)}{\Gamma \blacktriangleright \textbf{translate } SP \textbf{ by } s \Rightarrow P(s(\Sigma))}$$

$$\frac{\Gamma \blacktriangleright SP \Rightarrow P(\Sigma) \qquad \Gamma, \Sigma \blacktriangleright SP' \Rightarrow P(\Sigma')}{\Gamma \blacktriangleright \textbf{enrich } SP \textbf{ with } SP' \Rightarrow P(\Sigma \cup \Sigma')}$$

Figure 7.3: Rough typing programs and ASL terms

$$\frac{\blacktriangleright \Gamma \qquad \Gamma(X) \equiv \Sigma}{\Gamma \blacktriangleright X \Rightarrow \Sigma/X}$$

$$\frac{\blacktriangleright \Gamma \qquad \Gamma(X) \equiv \tau \qquad \tau \text{ non-signature}}{\Gamma \blacktriangleright X \Rightarrow \tau}$$

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

$$\frac{\begin{array}{ll} \Gamma \blacktriangleright M \Rightarrow \pi X : A.\tau' & \Gamma \blacktriangleright X \Rightarrow \tau_0 \\ \Gamma \blacktriangleright A \Rightarrow \mathsf{P}(\tau) & \Gamma \blacktriangleright \tau_0 \leq \tau \end{array}}{\Gamma \blacktriangleright M\, X \Rightarrow \tau'}$$

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

$$\frac{\Gamma \blacktriangleright A \Rightarrow \mathsf{P}(\tau)}{\Gamma \blacktriangleright \mathsf{Spec}(A) \Rightarrow \mathsf{P}(\mathsf{P}(\tau))}$$

$$\frac{\begin{array}{ll} & \Gamma \blacktriangleright A \Rightarrow \mathsf{P}(\tau) \\ \Gamma \blacktriangleright M \Rightarrow \tau_M & \exists \Sigma.\, \tau \equiv \Sigma \bigvee \tau \equiv \mathsf{P}(\Sigma) \\ \Gamma, X:\tau_M \blacktriangleright N \Rightarrow \tau_N & \Gamma, X:\tau_M \blacktriangleright \tau_N \leq \tau \end{array}}{\Gamma \blacktriangleright [X = M]N : A \Rightarrow \tau}$$

Figure 7.4: Rough typing ASL+ terms

$$\frac{}{\langle\rangle \;\Rightarrow\; \langle\rangle}$$

$$\frac{\Gamma \;\Rightarrow\; \Gamma_R \qquad \Gamma_R \blacktriangleright SP \;\Rightarrow\; P(\Sigma)}{\Gamma, SP \;\Rightarrow\; \Gamma_R, \Sigma}$$

$$\frac{\Gamma \;\Rightarrow\; \Gamma_R \qquad \Gamma_R \blacktriangleright A \;\Rightarrow\; P(\tau)}{\Gamma, X : A \;\Rightarrow\; \Gamma_R, X : \tau}$$

$$\frac{\Gamma \;\Rightarrow\; \Gamma_R \qquad \Gamma_R \blacktriangleright M \;\Rightarrow\; \tau}{\Gamma, X = M \;\Rightarrow\; \Gamma_R, X : \tau}$$

Figure 7.5: Rough typing full contexts

Theorem 7.7 (Strong normalization).
If $\Gamma \blacktriangleright M \;\Rightarrow\; \tau$ then M is β-strongly normalizing.

Proof The limitation on the kinds of β-redices that can appear means that a β-reduction reduces the number of redices in a term. Theorem 7.6 says that the result of reducing is a typable term, so reduction terminates. □

7.2.3 Using rough typing

Now I shall give an example of rough typing to demonstrate that it solves the two problems described in Section 5.8 — there is a useful notion of context, and desired propagation of type identities takes place.

To perform rough type-checking with respect to a "full" context, we must first extract a rough context from the full one. A full context type-checks to a rough context, written

$$\Gamma \;\Rightarrow\; \Gamma_R$$

by the rules shown in Figure 7.5. A practical implementation would probably store rough types along with the full terms in a single environment, rather than having two separate notions of context.

Now let's build up a rough context by type-checking a full one according to the rules in Figure 7.5. Actually, we only need the first and last of them. The first declaration is

```
ELT =sig
```

```
        type elt
    end
```

Let the denotation of this expression be Σ_{ELT}, so we have the rough typing:

$\langle\rangle \blacktriangleright \mathtt{ELT} \Rightarrow P(\Sigma_{ELT})$.

Now we declare a functor for building lists over some Σ_{ELT}-algebra:

```
List =λElt : ELT . alg
                     type elt = Elt.elt
                     type list = list_elt
                     val nil : list = ...
                     ⋮
                   end
```

($list_{\mathtt{elt}}$ is a complex type-expression in FPC which expresses the type of lists over the type `elt`). This has the rough typing

$\Gamma_1 \blacktriangleright \mathtt{List} \Rightarrow \Pi\mathtt{Elt}{:}\mathtt{ELT}.\Sigma_{LIST}[\mathtt{Elt.elt}]$

where $\Gamma_1 \equiv \mathtt{ELT} : P(\Sigma_{ELT})$, writing the inferred signature of the algebra as $\Sigma_{LIST}[\mathtt{Elt.elt}]$ to indicate the dependency on `Elt`. This signature contains the equation `elt = Elt.elt`.

Now let's apply the `List` functor to an algebra, declaring

```
Nat =alg
        type elt = nat
    end
```

Let Γ_2 be the rough context which extends Γ_1 with the declaration of `List` with its rough typing above, and Γ_3 the context which extends Γ_2 with the declaration of `Nat` with the type $\Sigma_{ELT}[nat]$ — this is the inferred type for `Nat` which expresses the equality between `elt` and *nat*.

Now we can derive:

$\Gamma_3 \blacktriangleright \mathtt{List\ Nat} \Rightarrow \Sigma_{LIST}[\mathtt{Nat.elt}]$

so if we extend the full context by the declaration

```
ListNat =List(Nat)
```

we have a corresponding rough context Γ_4 which declares `ListNat` with the rough type above. Then in the underlying FPC signature Σ_{Γ_4}, we can prove the equation `ListNat.elt` = *nat*. So the identity of the argument type has been successfully propagated to the result, and this solves the fundamental problem posed in Section 5.8.2.

7.3 Design for Modules in FPC

In the following subsections, I discuss some of the issues in the design of ASL+$_{\text{FPC}}$, to explain how the definitions in Section 7.2 were reached. This also explores the language a little bit further than the short example above. But readers uninterested in this entertaining interlude may skip to the semantics of the language, in Section 7.4 on page 250.

Constructing the institution $\mathcal{FPC}$ in the previous chapter was straightforward. Now we wish to add the higher-order module constructions and specification building operators of ASL+. Ideally, this should be a modular construction itself; we want to use the institution $\mathcal{FPC}$ (or another core-language) without modification, so we can be sure that properties of $\mathcal{FPC}$ are not disturbed by adding ASL+ "on top".

The idea that adding a module system to a language should be a modular construction itself has been promoted for the SML and EML module languages by Sannella [Sannella and Tarlecki, 1986, Sannella and Wallen, 1992], and followed by Leroy ín an implementation of one of his type systems [Leroy, 1996a].

In practice, the restriction that the core-language cannot be modified to add modules is too harsh. One small change is to add dot notation to the language to access module components.

7.3.1 Dot notation and contexts

There is a problem to confront immediately. Inside an FPC program, there is no way to refer to other programs — this is a serious difficulty for modular programming! For the abstract version of ASL+ in Chapter 5, I postulated program building operators which allow the extension of one program by some more declarations, for example. Here I shall take a more direct approach, using *dot notation* to project components from a module-language expression. For example, we could write $F(P).c$ to stand for the type component c of the module $F(P)$ and then use this within type expressions in another module. But expressions like $F(P)$ cannot occur in FPC programs, because that would imply a circularity: the need to define $\mathcal{FPC}$ at the same time as ASL+$_{\text{FPC}}$.

How can we break this circularity? The syntax and semantics of FPC types are defined with respect to a fixed signature. The signature does not mention names of variables used in the module-language, or their types. And the semantics of algebras provides no way of accessing environments for the module language.

A simplistic solution is to treat the use of dot-notation as a construction on *names* only, so that if $c \in \mathit{TyConst}$, X is a module-level variable, then $X.c$ is a new type constant. Since the syntax for algebras and signatures in Section 6.8.2 was presumptuously factored by a notion of *context*, this approach looks promising: we can build up a signature for the context which names all of the module expressions we need to use, using the dot notation to rename their signatures and put them together. Semantically, these syntactic operations construct a colimit in the category of signatures.

The difficulty with this simplistic solution is that the expression $X.c$ is just a name in FPC, and not amenable to substitution; $F(P).c$ still has no meaning, unless we enlarge $\mathit{TyConst}$ to include the *whole* syntax. That, apart from being unaesthetic, is not really feasible, since there is a context-sensitive equality on module language expressions. To prove as many type equalities as possible, we should be able to use $F(P) = F(P')$ to deduce $F(P).c = F(P').c$, so the module-level context is important. Again, it could not be accommodated without changing the definition of FPC signature.

Accepting that we cannot substitute module-level expressions into FPC types and terms, my solution is to push the simple variable-renaming approach as far as possible. Instead of writing a program:

```
alg
   type $c = F(P).d \times nat$
end
```

we must use a local binding to write:

```
local
   $X = F(P)$
in
   alg
      type $c = X.d \times nat$
   end
end
```

The local binding is part of the module language. One might imagine a syntactic translation which automatically lifts the use of module expressions outside core-level programs in this way, although in many cases it should not be too uncomfortable to write the long-hand directly. (This form is already a requirement in SML, for example.)

The difference with the abstract approach of PBOs and SBOs in Chapter 5 is that a connection *is* now made between the core-language context (FPC signature) and the module-level context: we can build the former directly from the latter, by picking out all the components which denote algebras.

7.3.2 Hiding the bound names

There is a question about the local-binding approach suggested above: what signature does the overall algebra expression have?

The algebra is defined in a context $\Sigma_{ctx} \cup X.\Sigma_X$ which includes the whole signature of $F(P)$. The signature Σ_{ctx} is the signature of the current context, Σ_X is the pre-signature of $F(P)$, and $X.\Sigma_X$ is a renamed copy of Σ_X obtained by prefixing all the component names by X. Outside the scope of the binding, though, we must remove mention of X from the signature.

Suppose the signature Σ_X of $F(P)$ is (the denotation of):

```
sig
   type d = nat
end
```

Then we can remove mention of X by replacing $X.d$ by *nat*, to get for the whole expression the signature:

```
sig
   type c = nat × nat
end
```

The type-checker takes advantage of the type equation in the signature of $F(P)$ to *propagate* the type equality and deduce that $c = nat \times nat$ in the result.

However, this is not possible in general, since there may be no type equation for d in the signature Σ_X. One could simply remove the sharing equation for c from the signature, but there would still be a difficulty if $X.c$ appeared in the type of a value rather than a sharing equation.

What we need is a general way to remove or hide the X part of the context in the result. One way is to make a *closure restriction*, to force the user to declare all the types used in a signature within that signature (this is the approach taken in several languages, including COLD-K [Jonkers, 1989] for example). This is useless unless we have *some* way to refer to the context, so at least sharing equations must contain open terms. But then the problem remains.

Another way to remove the parameter X is to automatically expand the signature with an extra type when the possibility of a signature without a name for a type arises. (Both this possibility and the last are part-way steps towards pushout parameterisation when the whole parameter is explicitly copied to the result; but we want to include something less than the whole parameter, and also find a syntactic method for handling identifiers.)

Following a syntactic approach and identifying program-level names with the names in the semantic signature, adding extra types to the signature leads to the tricky question of finding program-level identifiers for the extra types. In SML, the extra types do not need program-level identifiers: there is a distinction between names in semantic and syntactic signatures, and a semantic signature may have bound names for which there is no corresponding program-level identifier. Signatures with bound names occur in the elaboration of programs, but have no direct program-level syntax. With the limited module system of SML, this is not so much of a problem,[5] but with higher-order modules, it becomes useful to write signatures with bound names.

Instead of adding extra program identifiers, one of Leroy's module typing systems [Leroy, 1995] takes the opposite approach, by attempting to automatically remove dependencies and by deleting sharing equations. When this fails and a value occurs which has a type that cannot be expressed syntactically, that value is deleted from the result signature.

FPC signatures do not have hidden parts,[6] so my approach avoids bound names in signatures, like the syntactic module type systems for programming due to Leroy and others. Rather than use one of the slightly distasteful methods above to do this, I instead burden the user, and require the **local** construct to have a signature valid in the context without X. So we must write something like this:

```
local
   $X = F(P)$
in
   alg
      type $c = X.d \times nat$
   end : sig
            type $c$
         end
end
```

In this case the user has given the permission for the equation about the identity of c to be removed. Another possibility would be to introduce a new type d in the result algebra: then the user could express that c is equal to $d \times nat$.

In the formal syntax in Section 7.2, **local** is written more briefly as $[X = M]N : A$, and the signature is generalised to any specification term. Using

[5]except that some compilers invent strange type names like `?.t`, and then give even stranger error messages like `Type error:  ?.t does not match ?.t!`

[6]although perhaps they should; some proof methods for specifications with hidden parts rely on information about the structure of the signature [Farrés-Casals, 1992, for example].

a specification term has no difference for rough typing, but it is useful in the satisfaction system, to serve as a place to put axioms exported from the body, for example.

7.3.3 Making rough types dependent

In Chapter 5, I described the problem with using an arrow type of the form $\Sigma_1 \Rightarrow \Sigma_2$ for the type of a parameterised program: it doesn't express any relationship between Σ_1 and Σ_2. This means that the type-checker cannot deduce propagation of types from argument to result, and therefore cannot know when certain types which should be regarded as equal are in fact equal. This makes the notion of an environment or context unworkable, since there is no way to relate the results of building modular programs to their inputs.

The solution I have adopted is to introduce dependent products into rough types:

$$\tau \ ::= \ \Sigma \ \mid \ \pi X : \tau . \tau \ \mid \ P(\tau)$$

We have dependent types here because module-level variables X can occur (necessarily) free in the names of a pre-signature Σ.

For example, let:

$$\Sigma_c = \textbf{sig} \ \textbf{type}\ c \ \textbf{end} \qquad \Sigma_{cv[X.c]} = \textbf{sig}\ \textbf{type}\ c\ \ \textbf{sharing}\ c = X.c\ \ \textbf{val}\ v : c\ \textbf{end}$$

(strictly, there should be semantic braces around the syntax of the signatures above). I call the second signature $\Sigma_{cv[X.c]}$ to informally indicate the free name $X.c$ inside which shares with the type c. By this convention, the same signature without the sharing equation would be called Σ_{cv}. The signature might have other names $X.d, X.e$ appearing inside too; all these would be taken to be bound by the same module variable X.

The rough type $\pi X : \Sigma_c . \Sigma_{v[X.c]}$ describes a choice program which returns some element of its argument type. Suppose F is such a program, and we have an argument structure P = **alg type** $c = nat$ **end**, then the expected argument and result signatures of F would be:

$$\Sigma_{cn} = \textbf{sig}\ \textbf{type}\ c = nat\ \textbf{end} \qquad \Sigma_{v[nat]} = \textbf{sig}\ \textbf{type}\ c\ \ \textbf{sharing}\ c = nat\ \ \textbf{val}\ v : c\ \textbf{end}$$

(again, missing semantic braces).

The application of F is typed by the rule:

$$\frac{F \implies \pi X : \Sigma_c.\Sigma_{v[X.c]} \qquad P \implies \Sigma_{cn} \qquad \Sigma_c \subseteq \Sigma_{cn}}{F(P) \implies (\Sigma_{v[X.c]})[nat/X.c]}$$

the inferred signature Σ_{cn} expresses that c shares with nat. This sharing gets propagated to the result signature $(\Sigma_{cv[X.c]})[nat/X.c]$ which is equal to the signature $\Sigma_{cv[nat]}$ shown above.

So far so good; but we have the same difficulty as before with arguments for which no concrete (sharing) type is known, when the type c is *abstract*. Such arguments occur if there is some mechanism for providing abstraction in the module language (for example, by restricting the signature for P to one which conceals the implementation of c); or in any case, via specifications — instead of an explicit algebra, the argument could be a variable Y assumed to satisfy some specification.

Suppose the argument is a term M, where $M.c$ is an abstract type for one of these reasons. Then we cannot substitute $M.c$ for $X.c$ in the signature Σ_v. If M is a non-variable, then neither can we substitute M itself for X, since we cannot write $M.c$ inside FPC types. We could try solving this problem in the same way as for ASL+ terms in Section 7.3.1 above, by introducing a local binding for X around the rough type. This would lead to a further revised syntax for rough types,

$$\tau \ ::= \ \Sigma \ \mid \ \pi X : \tau.\tau \ \mid \ P(\tau) \ \mid \ [X = M]\tau$$

However, this would give a system with true term dependency (previously the dependency was just on names), and it is tricky to define an effective way of deciding when two such types are equal: a type-checking procedure would have to consider permutation of bindings around signature rough types, as well as term equality.

Instead I chose a pragmatic and approximate solution: restrict the application of parameterised programs to terms which we can always substitute into the body of a rough type. In this case, we can rename inside rough types, so we can have applications with the form $F\,X$. This is the only kind of application that is allowed in ASL+$_{\text{FPC}}$, although other kinds of application could be added. For example, we could allow application to an algebra expression P without hidden type identities, because we can always give a type to $F\,P$ by substituting the type definitions of P. (See also the discussion in Section 7.6.1 on page 257.)

The restriction on applications may seem limiting, but in practice one usually works with an environment of bindings to variables anyway, so the restriction is that the operand is something defined in the environment or a parameter declared in the context. Since there is a notation for declaring local bindings, this does not mean an ever-increasing environment. And using local bindings, the problem of removing the dependency on the operand has been passed on to the programmer.

To understand what happens with the $F(P)$ example above when P is a variable defined in the environment, see the example in Section 7.2.3.

7.3.4 Dependency between parameters

The rough types shown in the last section allow basic parameter-result sharing to be handled, but the scheme is unsound for some cases of higher-order parameterisation, because dependency between parameters is lost. (By "unsound" I mean that some terms are roughly-typable and yet do not appear in the satisfaction system; of course, we expect this because the satisfaction systems checks logical consequences, which can eliminate some roughly-typable terms. But we'd like the gap between the two type systems to be as small as possible, so we can eliminate the largest possible set of wrong terms.)

Here is an example of dependency between parameters. Consider a variation of the example before, where we have an ASL+ specification:

$$
\begin{array}{l}
\mathtt{CHOICEFUN} =_{\text{def}} \Pi X \leq (\textbf{spec} \\
\qquad\qquad\qquad\qquad \textbf{type}\ c \\
\qquad\qquad\qquad\qquad \textbf{axiom}\ \exists x : c.\, x \neq \bot_c \\
\qquad\qquad\qquad\quad \textbf{end})\ . \\
\qquad\qquad\quad \Pi Y : X\ .\ \textbf{sig} \\
\qquad\qquad\qquad\qquad\qquad \textbf{type}\ c = Y.c \\
\qquad\qquad\qquad\qquad\qquad \textbf{val}\ v : c \\
\qquad\qquad\qquad\qquad\quad \textbf{end}
\end{array}
$$

this has the rough type:

$$\pi X : P(\Sigma_c).\ \pi Y : \Sigma_c.\ \Sigma_{v[Y.c]}$$

which expresses that it is the type of functions mapping Σ_c-specifications to functions from Σ_c-algebras m to an algebra with a chosen element from $|m|_c$. This rough type has lost the dependency of the parameter Y upon the parameter X. If we apply a function F : CHOICEFUN to a specification for which some concrete implementation of c is given,[7]

$G =_{\text{def}} F$ (**spec type** $c = nat$ **end**)

[7]Notice that the argument to F is valid because the axiom $\exists x : c.x \neq \bot_c$ is satisfied once $c = nat$, although we wouldn't expect rough typing to detect this.

by the usual application rule, the rough type of G is

$$\pi Y : \Sigma_c.\Sigma_{v[Y.c]}$$

However, G should only be applicable to Σ_c-algebras in which $c = nat$, rather than any Σ_c-algebra. The rough type of G ought to be:

$$\pi Y : \Sigma_{c[nat]}.\ \Sigma_{v[Y.c]}$$

The problem is that specification parameters, which can be substituted with specifications having more sharing than their types require, may appear inside parameter specifications for algebras. With rough types of the form $\pi X : P(\tau).\pi Y : \tau'.\tau''$ any dependency of τ' on X has been lost. This problem is unique to the specification setting, and does not appear in the modular programming type systems studied by Leroy and others.

One fix could be to extend dot notation to allow projection on *specification* expressions, as well as algebra expressions. The rationale behind this is if a specification *SP* has a fixed implementation of some type c, which is expressed in its signature as sharing equation $c = nat$, for example, then any program P satisfying *SP* must have $P.c = nat$. In the example above, the rough type of G might be written $\pi Y : \Sigma_{c[X.c]}.\Sigma_{v[Y.c]}$. However, dot notation on specification expressions in other cases is not well-defined, because different models of a loose specification *SP* can implement c with different concrete types. So it is hard to see how to add it.

The solution to this problem I chose for ASL+$_{\text{FPC}}$ is to allow re-calculation of the rough type of the parameter, so that rough types become

$$\tau \ ::= \ \Sigma \quad | \quad \pi X : A.\tau \quad | \quad P(\tau)$$

where A is an ASL+ term. Now the rule for application must perform some substitution inside full terms A as well as in rough types[8], but whenever we need to compare rough types, we compare $\pi X : A.\tau$ and $\pi X : A'.\tau'$ by comparing the rough types of A and A'.

In the case above, the rough type of `CHOICEFUN` becomes:

$$\pi X : Spec(\textbf{sig type } c \ \ \textbf{axiom } \exists x : c.x \neq \bot_c \ \textbf{end}).\pi Y : X.\Sigma_{v[Y.c]}$$

and then the rough type of G would be

$$\pi Y : \textbf{sig type } c = nat \ \textbf{end}.\Sigma_{v[Y.c]}.$$

So now G is only applicable to algebras in which $c = nat$, as required.

[8]for type-checking, perhaps substitution of type expressions would be enough.

7.3.5 Type abstraction and hiding

In module systems for programming languages, *type abstraction* is achieved by *hiding type identities.* For example, if we define a stack module:

```
P =def alg
           type stack = list_nat
           val empty = ...
               ⋮
       end
```

Then giving this module the signature

```
S =def sig
           type stack
           val empty : stack
               ⋮
       end
```

prevents access to the underlying representation of *stack*, apart from through the interface functions. This is because the type-checker conceals the identity of *stack*, so elements of *stack* aren't amenable to the usual operations on $list_{nat}$.

There are different choices of how abstraction is introduced. If we have a "signature constraint" operation on modules as part of the language, then

$$P : S$$

is a module like P except with the identity of *stack* hidden.

With a *generative* semantics for this construct, each time the expression appears a fresh and distinct *stack* type would be generated, and so we would have that $P : S \neq P : S$. With an *applicative* semantics, we would consider all abstract types generated in the same way to be equal, so $P : S = P : S$. The argument in favour of a generative semantics is that it avoids "accidental" identification of types: if two abstract types happen to have the same implementation, they should not necessarily be considered equal, only types generated at the same point in the program should be considered equal. The argument against a generative semantics is that it is harder to reason about than an applicative semantics, because it does not admit the principle of substitution of equals-for-equals. For this reason, the treatment of ASL+ follows an applicative approach.

Semantically, the applicative interpretation for the constraint operation above is simply to apply the reduct functor $-|_{\Sigma_S} : \mathbf{Mod}(\Sigma_P) \to \mathbf{Mod}(\Sigma_S)$ to P, where Σ_P is the inferred signature for P which reveals the implementation of *stack* and Σ_S is the constraining signature which hides it. There is no constraint operation like this in ASL+$_{\text{FPC}}$ because it can be simulated using the binding construct, which has a constraining signature as explained in Section 7.3.2.

Coming to ASL-style specifications, on the other hand, type abstraction means more than just hiding type identities via the reduct functor. For example, the specification

$$\textbf{derive from}\ \{P\}\ \textbf{by}\ \iota : \Sigma_S \to \Sigma_P$$

has just one model, $[\![P]\!] \in \mathbf{Mod}(\Sigma_S)$. Although the identity of *stack* has been hidden in the signature of this specification, its class of models is unchanged by the **derive** operation. Instead, to give a specification of algebras which implement the stack operations, we would have to use the **abstract** operation of ASL. In fact, a more high-level specification language would probably interpret the ":" constraint operator appearing before specifications as a combination of **derive** and **abstract**.

This example of the difference between type abstraction in programming and specification (at least, in terms of the ASL+ semantics) raises questions over how to type-check the **derive** and **translate** operations. This we investigate next.

7.3.6 Handling renaming

The crux for propagating sharing properly is to allow the result signature of a parameterised program or specification to vary according the argument signature. Since parameterisation is understood in the syntax via substitution, we want to capture this variation in signature by a more liberal form of substitution, which allows a variable to substituted by a variable having a more refined type.

The refinement on types allowed is an "increase in sharing", so that if $\Sigma \subseteq \Sigma'$, but Σ' only differs from Σ in having more sharing equations, then we may substitute a Σ'-algebra for a Σ-algebra. Semantically in $\mathcal{FPC}$, we have an inclusion between $\mathbf{Mod}(\Sigma')$ and $\mathbf{Mod}(\Sigma)$ in this case, which makes it easier to deal with. (Easier, that is, than the more flexible case that Σ' also contains additional sorts and operations over Σ; comments about this alternative are made later.)

Using rough types we want express, as accurately as possible, the propagation behaviour of terms in the module language. For variables, we use the idea of *strengthening* the type from the context, to reflect that it shares with the context.[9] The typing rules for explicitly given algebras simply copy the type expressions into the signature (this happens in the type-checking rules in Section 6.8.2). We briefly considered functions above. What's left to look at is the operators of ASL. Here we examine the renaming and hiding operators **translate** and **derive**.

Renaming with translate

The usual rule for rough typing **translate** is:

$$\frac{\blacktriangleright\ SP\ \Longrightarrow\ \mathsf{P}(\Sigma_1) \qquad \sigma : \Sigma_1 \to \Sigma_2}{\blacktriangleright\ \textbf{translate}\ SP\ \textbf{by}\ \sigma\ \Longrightarrow\ \mathsf{P}(\Sigma_2)}$$

In the semantics, we think of **translate** being interpreted by a family of operators T_{Σ_1,Σ_2}; this was the abstract understanding of SBOs used in Chapter 5.

However, any model of **translate** *SP* **by** σ has a σ-reduct in *SP*, which implies that whenever $c =_{\Sigma_1} t$, then it is safe to assume $\sigma(c) =_{\Sigma_2} \sigma(t)$. This would result in a more powerful rough type-checking system, since more terms with no denotation will be eliminated by rough typing.

For example, we expect that

$$\begin{array}{l} \Gamma\ \blacktriangleright\ \textbf{translate sig type}\ c = nat\ \textbf{end by}\ [c \mapsto d] : \Sigma_c \to \Sigma_d \\ \Longrightarrow \quad \mathsf{P}(\Sigma_d[d = nat]) \end{array}$$

where Σ_c and Σ_d each declare one unspecified type, and $\Sigma_d[d = nat]$ is the denotation of **sig type** $d = nat$ **end**.

Because signatures are allowed to vary, and because a function can be applied to a term with a more refined type than its apparent domain, *Sig*(*SP*) may have more sharing than the fixed domain Σ_1 of the signature morphism σ. So the sharing equation $c = t$ actually appears in a super-signature of Σ_1. Generalising, we get a rule like this:

$$\frac{SP\ \Longrightarrow\ \mathsf{P}(\Sigma_1') \qquad \sigma : \Sigma_1 \to \Sigma_2 \qquad \Sigma_1 \subseteq_{\mathsf{sh}} \Sigma_1' \qquad \Sigma_2' = \sigma^*(\Sigma_1')}{\textbf{translate}\ SP\ \textbf{by}\ \sigma\ \Longrightarrow\ \mathsf{P}(\Sigma_2')}$$

where $\sigma^*(\Sigma_1')$ denotes the signature $(Ty^{\Sigma_2}, Sh, Tm^{\Sigma_2})$ with the sharing equations *Sh* defined by

$$Sh(c) = Sh^{\Sigma_2}(c) \cup \left\{ \sigma(t_1) \;\middle|\; \exists c_1.\, \sigma(c) = c_1 \ \bigwedge\ c_1 = t_1 \in \Sigma_1' \right\}.$$

The signature Σ_2 has the (extra) sharing equations propagated from Σ_1' by σ. This constructs a pushout in $\mathbf{Sign}^{\mathcal{FPC}}$.

[9]See Definition 7.2. This terminology is due to Leroy [1994] and doesn't relate to the usual type-theoretic notion of "strengthening" a context by removing variables.

But there is an easier institution-*dependent* way to arrive at the same result. Suppose instead that the **translate** specification is built using the *syntax* for a signature morphism. This is just a set of renamings, so the same syntax can denote many morphisms between similar signatures. In particular, given a source signature Σ_1', the target signature determined by a renaming $s(\Sigma_2')$ will contain exactly the sharing equations required above. This results in a rule for **translate** which is more realistic and more readable:

$$\frac{SP \implies P(\Sigma) \qquad \blacktriangleright\ s \implies \Sigma \to s(\Sigma)}{\textbf{translate}\ SP\ \textbf{by}\ s \implies P(s(\Sigma))}$$

It relies on the "natural polymorphism" of the syntax.

Hiding with derive

The **derive** operator is used to construct specifications with hidden parts. In many institutions, hiding is necessary to express certain specifications. With FPC, we could perhaps do without hiding of operators, since we can write formulae which are existentially-quantified on terms of any order. But there is no way to write formulae which are existentially quantified upon types.

The usual rule for typing **derive** is this:

$$\frac{SP \implies P(\Sigma_2) \qquad \sigma : \Sigma_1 \to \Sigma_2}{\textbf{derive from}\ SP\ \textbf{by}\ \sigma \implies P(\Sigma_1)}$$

Let's consider whether any type equalities should be propagated between Σ_2 and Σ_1, particularly in case the argument specification *SP* has a supersignature of Σ_2 with more sharing. We expect the typing of **derive** to hide names as usual. The question of whether **derive** should conceal type *identities* was mentioned in Section 7.3.5. Concealing type identities in programs makes good sense because it is how type abstraction is achieved, but in specifications type abstraction is achieved with the **abstract** operator (which I won't consider here).

It would be possible to allow **derive** to propagate *some* type identities: those which can still be expressed in the result signature, for example. But this would be a half-solution. Again, I stick with the original scheme; to begin with, it is questionable whether we want the raw **derive** operator for arbitrary signature morphisms in the language — see the discussion on page 258.

In the end, the rule above is modified only to use concrete syntax for signature morphisms. As before, compared to a rule containing a fixed semantic signature morphism, this allows specifications over supersignatures of Σ_2 as arguments:

$$\frac{\Gamma \blacktriangleright SP \implies P(\Sigma_2) \qquad \Sigma_{ctx} \blacktriangleright S \implies \Sigma_1 \qquad \blacktriangleright\ s \implies \Sigma_1 \to \Sigma_2}{\blacktriangleright\ \textbf{derive from}\ SP\ \textbf{by}\ s : S \implies P(\Sigma_1)}$$

Because the destination of a signature morphism cannot uniquely determine its source, we must also give the syntax for a signature as part of the **derive** construct. (A common and convenient addition would be to use an operator **hide** to hide some specified sorts and constants from a signature.)

7.3.7 Redundancy of singletons

Because FPC has type equations built into signatures, a choice motivated in Chapter 5, the prototypical use of singleton types to construct specifications from algebras is redundant. The specification

$$\{X\}$$

where X denotes a Σ-algebra, can be expanded into this specification in the context declaring X:

```
spec
   type $c_i$
   sharing $c_i = X.c_i$
   val $v_i : t_i$
   axiom $v_i = X.v_i$
end
```

(the subscripting suggests several statements of each form, according to the types and values in Σ). Rather than explaining dot notation via singleton types as suggested by Sannella et al. [1990] (see Section 2.1.5), expressions like this now have a direct semantics, because we the underlying context is available explicitly as an algebra.

Type sharing equations are more expressive than singleton types with renamings. For example, suppose we have a signature for types with a default value:

```
$\Sigma_{cv}$ =def sig
              type $c$
              val $v : c$
           end
```

Then we can write the specification of a parameterised program which makes products of the Σ_{cv} type as

```
PRODFUN =def $\Pi X$ :sig
                        type $c$
                        val $v : c$
                     end . spec
```

```
        type $c = X.c \times X.c$
        val $v : c$
        axiom $v = \langle X.v, X.v \rangle$
    end
```

Now if Prod : PRODFUN, then Prod has the rough type

$$\pi X : \Sigma_{cv}.\ \Sigma_{cv} \cup \{\, c := X.c \times X.c \,\},$$

which allows the type-checker to know that the result of a Prod(P) is a Σ_{cv}-algebra whose type c is equal to $P.c \times P.c$. Using the singleton construct in place of sharing equations, this could not be expressed, because there is no (ordinary) signature morphism which can express the equation $c = X.c \times X.c$.

Because the primary reason for singleton types is no longer necessary, they are not included in the syntax of ASL+$_{\mathrm{FPC}}$.

7.4 Semantics

A set-theoretic semantics of the rough typing judgements is given in Definition 7.8 below. For simplicity, sets are used to interpret specifications, rather than PERs as in previous chapters. This means that the satisfaction system will not be sound for a contravariant rule for Π, and it must use the simpler equal-domains rule shown in Section 4.9.2.

The interpretation is a partial definition, because function applications can be undefined if the operand does not meet the semantic requirements of the function parameter. In fact, even if rough types were used as the parameters, the interpretation of a term might not be a well-defined element of the interpretation of its rough type; a rough type can be empty, because signatures can contain inconsistent sharing equations. This is in contrast to the semantics based on rough typing in Chapter 5, where rough types always have non-error elements, so if there are only rough types in parameters (or if all PBOs and SBOs are total), then roughly typed terms always have a defined (non-$\diamond$) interpretation. For simplicity, I do not formalize $\diamond$ in this semantics.

As before, a full soundness proof must wait for the satisfaction system, although soundness for rough typing, modulo definedness, is proved as a property of the definition below in Proposition 7.10.

The semantics exploits the fact that in $\mathcal{FPC}$, if $\Sigma \subseteq_{\mathsf{sh}} \Sigma'$, then whenever $m \in \mathbf{Mod}(\Sigma')$, then $m \in \mathbf{Mod}(\Sigma)$ too. This means, for example, that no coercion is needed to interpret the variable rule, although the signature of the obvious extension of $Alg(\gamma)$ is $\Sigma_\Gamma \cup \Sigma$ rather than $\Sigma_\Gamma \cup \Sigma / X$. The semantic inclusion means that no coercion functions are needed to interpret syntactic subtyping, and so no interpretation of the subtyping judgement is given below. If arbitrary subsignatures were allowed in the rough subtyping relation, this issue (and some others) would have to be tackled, see Section 7.6.1 for discussion.

The interpretation of the judgement $\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma$ is a function for extracting the underlying algebra of an environment. Applied to an environment γ, this function is written $Alg^\Gamma(\gamma)$ where γ is in the interpretation of the context Γ. The interpretation of the context Γ is written $\mathcal{M}[\![\blacktriangleright \Gamma]\!]_\gamma$. The interpretation of well-formed rough types is constructed beginning from $\mathcal{M}[\![\Gamma \blacktriangleright \Sigma]\!]_\gamma$, which is the set of algebras from $\mathbf{Mod}(\Sigma_\Gamma \cup \Sigma)$ which extend the algebra given by $Alg(\gamma)$. Finally the interpretation of a term $\mathcal{M}[\![\Gamma \blacktriangleright M \Longrightarrow \tau]\!]_\gamma$ is an element of $\mathcal{M}[\![\Gamma \blacktriangleright \tau]\!]_\gamma$, when defined.

Definition 7.8 (Interpretation of rough typing judgements).
The interpretation of each rough typing judgement is defined by induction over its derivation. Each clause in the definitions below corresponds to a rule in Figures 7.1–7.4. Meta-variables introduced wildly below correspond to ones appearing in the premises of the rules.

Algebra of an environment:

- $Alg^{\langle\rangle}(\star) = (\emptyset, \emptyset)$ (the unique object of $\mathbf{Mod}(\emptyset)$)
- $Alg^{\Gamma,\Sigma}((\gamma, m)) = m$
- $Alg^{\Gamma,X:\tau}((\gamma, m)) = \begin{cases} m & \text{if } \tau \equiv \Sigma, \\ Alg^\Gamma(\gamma) & \text{if } \tau \text{ is a non-signature.} \end{cases}$

Interpretation of rough contexts:

- $\mathcal{M}[\![\blacktriangleright \langle\rangle]\!] = \{\star\}$
- $\mathcal{M}[\![\blacktriangleright \Gamma, \Sigma]\!] = \left\{ (\gamma, m) \;\middle|\; \gamma \in \mathcal{M}[\![\blacktriangleright \Gamma]\!],\;\; m \in \mathcal{M}[\![\Gamma \blacktriangleright \Sigma]\!]_\gamma \right\}$
- $\mathcal{M}[\![\blacktriangleright \Gamma, X : \tau]\!] = \left\{ (\gamma, m) \;\middle|\; \gamma \in \mathcal{M}[\![\blacktriangleright \Gamma]\!],\;\; m \in \mathcal{M}[\![\Gamma \blacktriangleright \tau]\!]_\gamma \right\}$

Interpretation of rough types:

- $\mathcal{M}[\![\Gamma \blacktriangleright \Sigma]\!]_\gamma = \left\{ m \in \mathbf{Mod}^{\mathcal{FPC}}(\Sigma_\Gamma \cup \Sigma) \;\middle|\; m|_{\Sigma_\Gamma} = Alg^\Gamma(\gamma) \right\}$
- $\mathcal{M}[\![\Gamma \blacktriangleright \pi X : A.\tau']\!]_\gamma = \Pi_{m \in \mathcal{M}[\![\Gamma \blacktriangleright A \Rightarrow P(\tau)]\!]_\gamma} \mathcal{M}[\![\Gamma, X : \tau \blacktriangleright \tau']\!]_{(\gamma, m)}$
- $\mathcal{M}[\![\Gamma \blacktriangleright P(\tau)]\!]_\gamma = Pow(\mathcal{M}[\![\Gamma \blacktriangleright \tau]\!]_\gamma)$

Interpretation of ASL terms:

- $\mathcal{M}[\![\Gamma \blacktriangleright S \Rightarrow P(\Sigma)]\!]_\gamma =$
  $$\left\{ m \in \mathbf{Mod}(\Sigma_\Gamma \cup \Sigma) \;\middle|\; m|_{\Sigma_\Gamma} = Alg^\Gamma(\gamma) \right\}$$
- $\mathcal{M}[\![\Gamma \blacktriangleright P \Rightarrow \Sigma]\!]_\gamma = \mathcal{P}[\![\Sigma_{ctx} \blacktriangleright P \Rightarrow \Sigma]\!]_{Alg^\Gamma(\gamma)}$
- $\mathcal{M}[\![\Gamma \blacktriangleright \mathbf{impose}\ \varphi\ \mathbf{on}\ SP \Rightarrow P(\Sigma)]\!]_\gamma =$
  $$\left\{ m \in \mathcal{M}[\![\Gamma \blacktriangleright SP \Rightarrow P(\Sigma)]\!]_\gamma \;\middle|\; m \models^{\mathcal{FPC}}_{\Sigma_\Gamma \cup \Sigma} \varphi \right\}$$
- $\mathcal{M}[\![\Gamma \blacktriangleright \mathbf{derive\ from}\ SP\ \mathbf{by}\ s : S \Rightarrow P(\Sigma)]\!]_\gamma =$
  $$\left\{ m|_\sigma \;\middle|\; m \in \mathcal{M}[\![\Gamma \blacktriangleright SP \Rightarrow P(\Sigma')]\!]_\gamma \right\}$$
  where σ is the denotation of $\Sigma_\Gamma \blacktriangleright s \Rightarrow \Sigma \to \Sigma'$.
- $\mathcal{M}[\![\Gamma \blacktriangleright \mathbf{translate}\ SP\ \mathbf{by}\ s \Rightarrow P(s(\Sigma))]\!]_\gamma =$
  $$\left\{ m \in \mathbf{Mod}(\Sigma_\Gamma \cup s(\Sigma)) \;\middle|\; m|_\sigma \in \mathcal{M}[\![\Gamma \blacktriangleright SP \Rightarrow \Sigma]\!]_\gamma \right\}$$
  where σ is the denotation of $\Sigma_\Gamma \blacktriangleright s \Rightarrow \Sigma \to s(\Sigma)$.
- $\mathcal{M}[\![\Gamma \blacktriangleright \mathbf{enrich}\ SP\ \mathbf{with}\ SP' \Rightarrow P(\Sigma \cup \Sigma')]\!]_\gamma =$
  $$\left\{ m \in \mathbf{Mod}(\Sigma_\Gamma \cup \Sigma \cup \Sigma') \;\middle|\; \begin{array}{l} m|_{\Sigma_\Gamma \cup \Sigma} \in \mathcal{M}[\![\Gamma \blacktriangleright SP \Rightarrow P(\Sigma)]\!]_\gamma \wedge \\ m \in \mathcal{M}[\![\Gamma, \Sigma \blacktriangleright SP' \Rightarrow P(\Sigma')]\!]_{(\gamma, m|_{\Sigma_\Gamma \cup \Sigma})} \end{array} \right\}$$

Interpretation of ASL+ terms:

- $\mathcal{M}[\![\Gamma \blacktriangleright X \Rightarrow \Sigma/X]\!]_\gamma = (\iota_X, \rho_X)$ given by
  $$\iota_X(c) =_{\text{def}} \begin{cases} \iota_\Gamma(c) & c \in \Sigma_\Gamma, \\ \iota_\Gamma(X.c) & c \in \Sigma. \end{cases} \qquad \rho_X(v) =_{\text{def}} \begin{cases} \rho_\Gamma(v) & v \in \Sigma_\Gamma, \\ \rho_\Gamma(X.v) & v \in \Sigma. \end{cases}$$
  where $(\iota_\Gamma, \rho_\Gamma) = Alg^\Gamma(\gamma)$.
- $\mathcal{M}[\![\Gamma \blacktriangleright X \Rightarrow \tau]\!]_\gamma = \gamma^\Gamma(X)$
- $\mathcal{M}[\![\Gamma \blacktriangleright \lambda X{:}A.\,M \Rightarrow \pi X : A.\tau']\!]_\gamma =$ the function f given by
  $$f(m) = \mathcal{M}[\![\Gamma, X : \tau \blacktriangleright M \Rightarrow \tau']\!]_{(\gamma, m)}$$
  for $m \in \mathcal{M}[\![\Gamma \blacktriangleright A \Rightarrow P(\tau)]\!]_\gamma$.
- $\mathcal{M}[\![\Gamma \blacktriangleright M\,X \Rightarrow \tau']\!]_\gamma =$
  $$\mathcal{M}[\![\Gamma \blacktriangleright M \Rightarrow \pi X : A.\tau']\!]_\gamma\,(\mathcal{M}[\![\Gamma \blacktriangleright X \Rightarrow \tau_0]\!]_\gamma)$$
- $\mathcal{M}[\![\Gamma \blacktriangleright \Pi X{:}A.\,B \Rightarrow P(\pi X : A.\tau')]\!]_\gamma =$
  $$\Pi_{m \in \mathcal{M}[\![\Gamma \blacktriangleright A \Rightarrow P(\tau)]\!]_\gamma} \mathcal{M}[\![\Gamma \blacktriangleright B \Rightarrow P(\tau')]\!]_{(\gamma, m)}$$

- $\mathcal{M}[\![\Gamma \blacktriangleright \mathsf{Spec}(A) \implies \mathsf{P}(\mathsf{P}(\tau))]\!]_\gamma = Pow(\mathcal{M}[\![\Gamma \blacktriangleright A \implies \mathsf{P}(\tau)]\!]_\gamma)$
- $\mathcal{M}[\![\Gamma \blacktriangleright [X = M]N : A \implies \tau]\!]_\gamma = R_\tau(k)$ where

$$k =_{\text{def}} \mathcal{M}[\![\Gamma, X : \tau_M \blacktriangleright N \implies \tau_N]\!]_{(\gamma, m)}$$

  where $m = \mathcal{M}[\![\Gamma \blacktriangleright M \implies \tau_M]\!]_\gamma$, and

$$R_\tau(k) = \begin{cases} k|_{\Sigma_\Gamma \cup \Sigma} & \text{if } \tau \equiv \Sigma \\ \{ a|_{\Sigma_\Gamma \cup \Sigma} \mid a \in k \} & \text{if } \tau \equiv \mathsf{P}(\Sigma) \end{cases}$$

  where $\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma$. □

We shall use the full contexts in the next section; these have an obvious interpretation using the definition above, by induction on a derivation of $\Gamma \implies \Gamma_R$.

Definition 7.9 (Interpretation of contexts).
The interpretation of a context Γ such that $\Gamma \implies \Gamma_R$ for some Γ_R is given by:

- $\mathcal{M}[\![\langle\rangle \implies \langle\rangle]\!] = \{ \star \}$
- $\mathcal{M}[\![\Gamma, SP \implies \Gamma_R, \Sigma]\!] =$
  $\left\{ (\gamma, m) \;\middle|\; \gamma \in \mathcal{M}[\![\Gamma \implies \Gamma_R]\!],\; m \in \mathcal{M}[\![\Gamma_R \blacktriangleright SP \implies \mathsf{P}(\Sigma)]\!]_\gamma \right\}$
- $\mathcal{M}[\![\Gamma, X : A \implies \Gamma_R, X : \tau]\!] =$
  $\left\{ (\gamma, m) \;\middle|\; \gamma \in \mathcal{M}[\![\Gamma \implies \Gamma_R]\!],\; m \in \mathcal{M}[\![\Gamma_R \blacktriangleright A \implies \mathsf{P}(\tau)]\!]_\gamma \right\}$
- $\mathcal{M}[\![\Gamma, X = M \implies \Gamma_R, X : \tau]\!] =$
  $\left\{ (\gamma, m) \;\middle|\; \gamma \in \mathcal{M}[\![\Gamma \implies \Gamma_R]\!],\; m = \mathcal{M}[\![\Gamma_R \blacktriangleright M \implies \tau]\!] \right\}$

Proposition 7.10 (Rough soundness).
Suppose $\Gamma \blacktriangleright M \implies \tau$. Then $\mathcal{M}[\![M]\!]_\gamma \in \mathcal{M}[\![\tau]\!]_\gamma$ whenever both are defined, for all $\gamma \in \mathcal{M}[\![\Gamma]\!]$, whenever $\mathcal{M}[\![\Gamma]\!]$ is defined.

Proof (Sketch). By induction on the typing derivation for M and formation derivation for τ. □

7.5 Satisfaction System for ASL+$_{FPC}$

Now we have an improved rough typing system which puts modules together in a context. We want to extend these changes to the satisfaction system. Here I shall just sketch a few of the rules a satisfaction system might contain. The rules would be similar to those presented for the abstract version of ASL+, in Sections 5.5 and 5.6.

There are four judgements used in the satisfaction system:

$\rhd \Gamma$	Γ is a well-formed context
$\Gamma \rhd M : A$	M satisfies A
$\Gamma \rhd M = N$	M and N are equal in context Γ
$\Gamma; M \vdash \varphi$	φ holds in M
$\Gamma \vdash \varphi$	φ holds in Γ

The first three judgements are as usual, except that equality judgement is not relative to a type here, because we have a set-theoretic semantics. The last two judgements prove properties of terms M or of the context Γ. In the first case, the property φ holds for the term M, which must denote an algebra or a set of algebras. In the second case, φ holds for the underlying algebra of the context Γ.

These new judgements allow combinations of properties to be proved for the context and for terms, and then used in rules which are certain kinds of "cut". As with the system presented in Chapter 5, there is no general logical rule of cut, since the satisfaction judgement is not a consequence relation which allows deduction from arbitrary premises. Instead properties of the context or terms can be used in key places. For example, a possible rule for **impose** would be:

$$\frac{\Gamma \rhd M : SP \qquad \begin{array}{l}\Gamma \Rightarrow \Gamma_R \\ \Gamma_R \Rightarrow_{\mathsf{sig}} \Sigma_\Gamma \\ \Gamma_R \blacktriangleright M \Rightarrow \Sigma\end{array} \qquad \begin{array}{l}\Gamma \vdash \varphi_\Gamma \\ \Gamma; M \vdash \varphi_M\end{array} \qquad \varphi_\Gamma \wedge \varphi_M \vdash^{\Sigma_\Gamma \cup \Sigma} \varphi}{\Gamma \rhd M : \textbf{impose } \varphi \textbf{ on } SP}$$

To show that M satisfies **impose** φ **on** SP, we must show that it satisfies SP and additionally that it satisfies φ. The logical satisfaction is proved in the signature which extends the signature of the context by the signature of M. To prove it, we may use any properties of the context and of M itself.

Properties of a term M can be derived in two ways. If M is an explicit program P, then we can assert an equality which comes from a declaration inside P:

$$\frac{\textbf{val } v : t = e \text{ appears in } P}{\Gamma; P \vdash v = e}$$

If on the other hand, M can be shown to satisfy or refine an **impose** specification already, we can assert the axiom directly:

$$\frac{\Gamma \rhd M : \textbf{impose}\ \varphi\ \textbf{on}\ SP}{\Gamma; M \vdash \varphi} \quad \frac{\Gamma \rhd SP : \mathit{Spec}(\textbf{impose}\ \varphi\ \textbf{on}\ SP')}{\Gamma; SP \vdash \varphi}$$

For ASL specifications, these rules rely on the presence of rules for reasoning about the equality of specification terms and proving refinements, to obtain a specification with **impose** as the outermost constructor.[10] Suitable rules can be formed by modifying those given in Wirsing [1990], Farrés-Casals [1992], for example.

For specifications built using parameterised terms, for example when $M = F\ X$, we must find a Π-type of the form $\Pi Y{:}A.\,\textbf{impose}\ \varphi\ \textbf{on}\ SP$ for F. This can arise from a *generic* proof for F, which relies on proving first $F : \Pi Y{:}A.\,\textbf{impose}\ \varphi\ \textbf{on}\ SP$ and then $X : A$. Or it can arise from a slightly different information flow: find an A such that $X : A$ and A is chosen to express enough information about X so that $F : \Pi Y{:}A.\,\textbf{impose}\ \varphi\ \textbf{on}\ SP$ can be derived. Both these techniques are possible, in particular because of the generalised λ-rule which we adopt (called (λ-NARROW) in Section 3.3.1 on page 54):

$$\frac{\Gamma \rhd A' : \mathit{Spec}(A) \qquad \Gamma, X : A' \rhd M : B' \qquad \Gamma, X : A \rhd M : B}{\Gamma \rhd \lambda X{:}A.\,M : \Pi X{:}A'.\,B'}$$

(Because we do not have the contravariant rule here, the use of this rule allows unrelated types to be derived for the same λ-term.)

Properties of the context are proved by renaming properties of terms inside the context. There are three rules to deal with the three different kinds of declaration a context may contain.

$$\frac{\Gamma \equiv \Gamma_1, X : SP, \Gamma_1' \qquad \begin{array}{l}\Gamma \rhd SP : \mathit{Spec}(\textbf{impose}\ \varphi\ \textbf{on}\ SP') \\ \Gamma_R \blacktriangleright SP \implies \mathit{Spec}(\Sigma)\end{array}}{\Gamma \vdash X_\Sigma.\varphi}$$

$$\frac{\Gamma \equiv \Gamma_1, X = M, \Gamma_1' \qquad \Gamma; M \vdash \varphi \qquad \Gamma_R \blacktriangleright M \implies \mathit{Spec}(\Sigma)}{\Gamma \vdash X_\Sigma.\varphi}$$

$$\frac{\Gamma \equiv \Gamma_1, SP, \Gamma_1' \qquad \Gamma \rhd SP : \mathit{Spec}(\textbf{impose}\ \varphi\ \textbf{on}\ SP')}{\Gamma \vdash \varphi}$$

Other rules for dealing with assertions allow properties φ to be combined, and the trivial derivation of T from any context or term.

[10] An alternative strategy would be to extend the definition of $\Gamma; M \vdash \varphi$ to allow theorem proving by decomposing the structure of M itself, as suggested for ASL by Sannella and Tarlecki [1988a].

As well as these new rules, we have rules similar to the ones seen before in λ_{Power} and $\lambda_{\leq\{\}}$ for the higher-order type constructors. To prove equalities, we also need congruence rules for the ASL terms, to admit equality relations on the syntactic representations of signatures and algebras, for example.

The rules suggested here are provisional; more work is needed to study their formal properties and suitability for the task. More rules are probably needed, and strategies for proof search should be studied — in practice one would wish for cunning ways to control the search space for proving consequences, as suggested by Sannella and Burstall [1983].

We end by stating without proof the desired agreement and soundness properties of the satisfaction system.

Theorem 7.11 (Agreement with rough typing).

1. *If $\rhd \Gamma$ then for some Γ_R, $\Gamma \implies \Gamma_R$.*
2. *If $\Gamma \rhd M : A$ then for some τ, $\Gamma_R \blacktriangleright M \implies \tau$ and $\Gamma_R \blacktriangleright A \implies \mathsf{P}(\tau)$, where $\Gamma \implies \Gamma_R$.*

Theorem 7.12 (Soundness).
Suppose $\rhd \Gamma$ and let $\gamma \in \mathcal{M}[\![\Gamma \implies \Gamma_R]\!]$.

1. *If $\Gamma \rhd M : A$, then $\mathcal{M}[\![M]\!]_\gamma \in \mathcal{M}[\![A]\!]_\gamma$.*
2. *If $\Gamma \rhd M = N$ and for some A, $\Gamma \rhd M : A$ and $\Gamma \rhd N : A$, then $\mathcal{M}[\![M]\!]_\gamma = \mathcal{M}[\![N]\!]_\gamma$.*
3. *If $\Gamma; M \vdash \varphi$ then for some Σ, $\Gamma_R \blacktriangleright M \implies \Sigma$ or $\Gamma_R \blacktriangleright M \implies \mathsf{P}(\Sigma)$ and $\mathcal{M}[\![M]\!]_\gamma \models^{\mathcal{FPC}}_{\Sigma_\Gamma \cup \Sigma} \varphi$.*
4. *If $\Gamma \vdash \varphi$ then $\gamma \models^{\mathcal{FPC}}_{\Sigma_\Gamma} \varphi$.*

In the second case, actually, the existence of A ought to follow from $\Gamma \rhd M = N$ by a meta-theorem of the satisfaction system.

7.6 Discussion

This section discusses some shortcomings and desirable improvements for the work here, and mentions some related work. The chapter is summarised in Section 7.7.

7.6.1 Shortcomings and improvements

The rough typing system presented here is only a first-step towards a more expressive type system, rather than a robust and final solution. Several problems with it remain, and there are some highly-desirable extensions. These are discussed below.

Propagation with higher-order modules

The examples in Chapter 2 made little use of higher-order modules: most of the modules used as parameters were ground terms. The type system presented here has some problems dealing with the expected propagation of sharing in more general cases.

The typical example[11] is the parameterised program ApplyFun which takes a parameterised program F as an argument and applies it to a second argument X. The application ApplyFun(F) should behave just as F itself.

$$\texttt{ApplyFun} =_{\text{def}} \lambda F : \textbf{sig type } c \textbf{ end} \rightarrow \textbf{sig type } c \textbf{ end} .\ \lambda X : \textbf{sig type } c \textbf{ end} .\ F(X)$$

We have the rough typing:

$$\langle\rangle \blacktriangleright \texttt{ApplyFun} \Rightarrow \pi F : \textbf{sig type } c \textbf{ end} \rightarrow \textbf{sig type } c \textbf{ end}.\ \pi X : \textbf{sig type } c \textbf{ end}.\textbf{sig type } c \textbf{ end}.$$

Now ApplyFun can be applied to a module which has a more refined type which expresses some sharing, for example,

$$\texttt{Id} =_{\text{def}} \lambda X : \textbf{sig type } c \textbf{ end} .\ \textbf{alg type } c = X.c \textbf{ end}$$

with

$$\langle\rangle \blacktriangleright \texttt{Id} \Rightarrow \pi X : \textbf{sig type } c \textbf{ end}.\Sigma_{c[X.c]}.$$

But the rough type of ApplyFun (Id) is

$$\langle\rangle \blacktriangleright \texttt{ApplyFun (Id)} \Rightarrow \pi X : \textbf{sig type } c \textbf{ end}.\Sigma_c$$

which has lost the sharing expressed by Id.

Leroy's syntactic solution to this [Leroy, 1995] is to introduce a new kind of name: as well as allowing names $X.c$, he considers $X(Y, \ldots).c$ to be a new name. Then the types of applications like ApplyFun above can be strengthened in the same way that variable types are, to express the sharing in the

[11] see Leroy [1995]

result as **type** $c = F(X).c$. (This means that the left hand sides of application terms have to be restricted too, although the right hand sides can now be applications of variables as well as just variables).

At this stage it is getting painful to maintain the restriction of not permitting module-level expressions inside core-level expressions. In retrospect, a better module language might be obtained by properly mixing the syntax; see the discussion in Section 8.2.

Better rough typing

The rough typing system does not guarantee that inconsistent signatures cannot arise during type-checking. One simple example is when **derive** is used to hide a type identity:

$$F =_{\text{def}} \lambda X{:}(\textbf{derive from } (\textbf{spec type } c = \mathit{bool}\ \textbf{end})\ \textbf{by } [c \mapsto c] : \Sigma_c).\, SP$$

Now the parameterised specification F is only applicable to classes of models in which c is implemented by *bool*, but this information has been hidden in the rough type of its parameter, which is Σ_c. Because we are allowed to pass parameters with *more* sharing, we can now write something like

$$F\,(\textbf{spec type } c = \mathit{nat}\ \textbf{end})$$

which is obviously wrong and would not be a term in the satisfaction system. But it is roughly typable. Other examples are possible which rely on the use of the binding construct, because that too can hide type-sharing.

What can we do to fix such problems? One approach may be to restrict the language: it isn't clear that having the full generality of the ASL operators available in a high-level language is right, and perhaps with more restrictive higher-level constructs such inconsistencies could be avoided. In the case above, a **derive** operation would not be allowed alone — instead we would have a combination of **derive** followed by **abstract**, which would allow different implementations of the type c once its identity was hidden.

Alternatively, the framework itself may be considered to be at fault. We have no distinction here between free names in signatures and "rigid" (or bound) names. Rigid names are those which have some fixed, but unknown, interpretation. This distinction is made in the SML stamp-based semantics, but the type-theoretic treatments of modular programming [Leroy, 1994, Harper and Lillibridge, 1994] manage to avoid it and treat types as I have done here: either "manifest" (with known sharing equations) or "abstract" (no known sharing equations, and free). The deep reason is that for modular programming, parameter signatures are fixed and there is no way to introduce rigid names in parameters. In ASL+ parameter signatures can vary because we have variables ranging over specifications which can be instantiated with specifications of a super-signature; and because of expressions like those above. This also explains why other studies have found parameterised signatures (and even parameterised renamings) to be useful.

Passing parameters with more components

In real languages for modular programming, one is usually allowed to substitute an algebra over any super-signature, so that extra components of the parameter are forgotten automatically. This is a richer form of subtyping than the one adopted here, but it is harder to cope with in the syntax and semantics, since it calls for the use of coercion functions (built up from the reduct functors), to "cut-down" an algebra to the right size, by forgetting extra components. To do this we would need to define an interpretation of the subtyping judgement as a coercion function, but more crucially, prove some kind of *coherence* property which relates the interpretation of the substitution of a coerced-term with the coercion of the substitution. See the remarks in Section 4.6 for similar discussion.

There is a further difficulty raised by allowing parameters with more components. The new components might obscure other declarations: in other words, it becomes possible to re-bind the same name. This is tricky to tackle; whilst names in signatures behave like bound variables, they cannot be renamed. Some form of distinction is needed between the names of the type and term constants in the signature, and the identifiers used in types and terms. This has been addressed in a couple of ways in the recent treatments of type-theoretic module systems (see for example Leroy [1994], Harper and Lillibridge [1994]).

Despite these difficulties, I believe that a proper treatment of modular programming and specification should allow this kind of subtyping.

Nested sub-algebras

In most module languages there is a facility for declaring a *sub-module* nested within another module. This is not allowed here; there are a couple of reasons why it is difficult to add.

The first reason concerns the mix of languages mentioned before: at the moment there is a strict separation between the core language FPC and the modular extension to ASL+. So the rules for writing syntactic signature and algebra expressions only have an FPC signature for a context, rather than a full ASL+$_{\mathrm{FPC}}$ context. This precludes directly including something like **mod** $L = M$ in an algebra, for example.

Instead we might try to extend the "variable-only" treatment, and allow the rules for signatures and algebras to understand the fact that some names have a special construction which indicates they denote components of a module. Then we might write **mod** $L = X$, for example.

However, in both cases there is a more serious problem to be addressed: again, the problem mentioned above of re-binding names. In this case, a nested module could have the same name as a module in the context — forcing it to be different seems rather too strong at this point.

The renaming by flattening approach to adding nested sub-algebras extends the treatment of signatures as contexts to signatures declared over those contexts. In other words, we would have extensions like $\Sigma_{ctx} \cup X.\Sigma \subseteq_{\mathsf{ext}\cup} \Sigma_1 \cup Y.\Sigma$. At this point, it becomes questionable whether flattening is right. In real implementations, the modular structure is retained: the dot-notation corresponds to a look-up operation rather than a fixed projection defined ahead of time based on a global renaming. Although it is tempting to explain nested sub-algebras by the flattening approach, I believe a better understanding would be gained by treating nested signatures properly. This would mean altering the underlying institution once again and not using $\mathcal{FPC}$ itself directly.

7.6.2 Related work on syntactic module type systems

Leroy, and Harper and Lillibridge independently, introduced the idea of signatures with explicit type-sharing equations to give a type-theoretic version of the SML module system which could explain the forms of type abstraction and propagation in SML [Harper and Lillibridge, 1994, Leroy, 1994]. The intention was to get an accurate and type-theoretic description which captures the operational description of the SML semantics Milner et al. [1990], and also repair problems with it such as the lack of support for separate compilation. Leroy [1995] later extended his work to consider higher-order modules.

The rough type system here follows some of these researchers' ideas, although it began life from the conception of adding type equations to algebraic signatures to explain sharing and fix the institutional semantics. Similar ideas for type equations in signatures were used by Tarlecki [1992]. I have aimed for a type system which relates directly to the abstract syntax of the module language; even some of Leroy's systems require extra stages of translation.

Many of the restrictions in the type system presented here arise from the inability to substitute module-level expressions into core-level expressions: the strict separation of the syntax of FPC from ASL+. This is why we introduce local bindings, which are not present in other systems. (In compensation, other systems have the possibility of nested modules, which can be hidden by a signature constraint operation.)

Leroy [1996b] makes restrictions on what kind of applications are allowed and where they may appear, similar to the restrictions made here because of the substitution problem. Leroy's motivation instead is to capture a *generative* semantics in which substitution is not a valid operation. A name-level understanding of type equality seems essential for dealing with a generative semantics.

Some recent work on module systems by Courant [1997a,b] attempts to remove these restrictions from Leroy's systems by introducing generation of type names at a different stage, when structure *declarations* are made ($X = F(M)$) instead of when functors are applied ($F(M)$); expressions then retain an applicative semantics, but declarations have to be treated specially. This idea might work well with the treatment of contexts here.

7.7 Summary

This chapter described an instance of ASL+ based on the institution $\mathcal{FPC}$ defined in Chapter 6. I introduced a new rough typing system for the language. This has two major improvements over the previous rough typing system for the abstract version of ASL+ in Chapter 5.

The first improvement given by the new type system is that it allows programs and specifications in-context to be dealt with, giving a formal treatment of dot notation for renaming the parameters in the bodies of abstractions, for example. Before this point, the dot notation was explained in an informal way by appealing to some translation on the source language involving singleton types (see the example in Section 2.1.5). This translation is less general than the present treatment which allows sharing equations which express sharing between *parts* of types.

The second improvement given by the new type system is that it explains the *propagation* of type information from the argument to the result in a parameterised program. This means that parameterised programs do not denote a single function between model classes over two fixed signatures $\Sigma_{arg} \rightarrow \Sigma_{res}$, but rather a *family* of functions indexed by signatures Σ'_{arg} such that $\Sigma_{arg} \subseteq_{\mathsf{sh}} \Sigma'_{arg}$, from the model class of Σ'_{arg} to the model class of a signature $\Sigma_{res}[\Sigma'_{arg}]$ which may depend on the sharing equations in Σ'_{arg}. This is closer to the situation in real programming languages and pushout parameterisation.

I gave a semantics for the language based on this new type system, which is an original step beyond the purely syntactic treatments of the work on programming languages. I agree with the arguments given by Leroy [1994], Harper and Lillibridge [1994] that there should be closer connection between the syntax and semantics than is evident in the *operational* explanations of type-checking (such as in the 1990 SML semantics Milner et al. [1990]) and explanations which are more type-theoretical are needed. However, the model-theoretic semantics is still paramount in the theory of algebraic specification, so it is important to connect the two sides. This poses some new challenges.

The rough typing system given here still has drawbacks, described in the previous section. It is an experimental first-step towards an expressive type system, rather than a robust and final solution. There is much scope for further research.

8 Conclusions

8.1 Summary

THE OBJECTIVE of this thesis was to study techniques and foundations for the modular development of large programs and specifications, using the language ASL+ which was introduced in the closing sections of Sannella et al. [1992].

The work began with two fundamental studies of type systems which are fragments of ASL+.

The first study was of the system $\lambda_{\leq\{\}}$, which combines subtyping with singleton types, introduced in Chapter 3. This was one of only a few studies of dependently-typed subtyping systems in the literature when it was first published [Aspinall, 1995a]. Since then more has been done and the research area is quite active, driven by a desire to add subtyping and similar constructs to theorem provers based on type theory [Aspinall and Compagnoni, 1996, Betarte and Tasistro, 1996, 1997, Bailey, 1996, Chen and Longo, 1996, Chen, 1997].

The second study was of the system λ_{Power}, introduced in Chapter 4. This appears to be the first in-depth study of power types in the literature. It forms the core of ASL+, and has a *rough typing system* which is decidable and approximates typing in the full system.

In Chapter 5, I gave a formalization of ASL+ based on the type systems given before. This involves special notions of consequence relations for proof systems of the underlying institution.

My goal was to capture ASL+ as an institution-independent *generic language*, which captures parameterisation as it is used in real specification and programming languages. In particular, I wanted to completely formalize the example of Chapter 2 and other similar examples. To get a useful language, we should just have to instantiate to a given institution and use the type-checking rules of ASL+, and not a complex translation and duplicated type-checking rules for higher-order modules in the source language. The original proposal of ASL+ falls short of this goal for technical reasons to do with type-checking, explained in Chapter 5.

With a view to solving these problems and formalizing the examples fully, in Chapter 6, I gave a complete description of an institution based on FPC, a minimal functional programming language with recursive types. The logic of the institution is an LCF-style extension to Higher-Order Logic.

In Chapter 7, I introduced a new rough typing system for $\mathrm{ASL+}_{\mathrm{FPC}}$, a version of ASL+ based on FPC. This system successfully captures the type-checking requirements of the examples in Chapter 2. However, there are similar examples involving use of higher-order parameterisation, for which the type-checking system of $\mathrm{ASL+}_{\mathrm{FPC}}$ is not sufficient.

It seems quite difficult to design a good type-checking system for ASL+ which deals properly with putting signatures together. In retrospect the difficulty isn't surprising; for the case of programming languages alone, research is still highly active into finding good understandings of higher-order module systems — the goal being to find understandings which are less operational and more type-theoretic [Leroy, 1994, 1995, 1996b,a, Harper and Lillibridge, 1994]. And for ASL+ we want to explain not only the process of type-checking, but also the model-theoretic semantics of the language.

The goal to find a suitable type-checking system is important because in a typical institution, the semantics of ASL+ is strongly typed, so type-checking is an absolute precursor to defining the semantics. It must also be a precursor to serious investigation of the proof system for proving satisfaction of a specification by a program, or refinement of specifications. The work here still leaves questions to be answered.

8.2 Future directions

Examples of program development

To begin with, further examples of formal development should be undertaken in languages like ASL+. The new forms of structuring programs with higher-order module languages are only now being explored, as such languages become available in prototype. ASL+ allows specifications to be structured in similar ways, and also allows study of the transition between specification structure and program structure, via Π-specifications. The importance of being able to specify program structure has been recognised in the on-going Common Framework Initiative [CoFI, 1997] into defining a paradigmatic algebraic specification language; the emerging language CASL provides *architectural specifications* which are akin to first-order Π-specifications with several arguments [CoFI Task Group on Language Design, 1997]. An important issue to look at is the possible development steps used in refining architectural specifications, comparing with the implementation relations used for parameterised specifications. These points were touched upon in Chapter 2, but more needs to be done.

Meta-theory of complex type systems

On the type theory side, the fundamental studies of Chapter 3 and Chapter 4 left some questions to be answered, particularly over the meta-theory of λ_{Power}. However, it is less clear whether λ_{Power} itself is necessary as a basis for ASL+, since the variety of parameterisation mechanisms possible in λ_{Power} have hardly been exploited in the ASL+ examples studied to date. So far, we have used algebras parameterised on algebras, specifications parameterised on specifications and specifications parameterised on algebras. As a type system, modelling specifications with types, this parameterisation structure is described by the corner of Barendregt's λ-cube known as $\lambda P\underline{\omega}$ [Barendregt, 1992]. For ASL+ we need a subtyping variant of $\lambda P\underline{\omega}$ which provides bounded abstraction and quantification. The particular system needed has not been studied in the literature yet, although work on extending other corners of the cube with subtyping comes close [Aspinall and Compagnoni, 1996, Chen, 1997, Compagnoni and Goguen, 1997].

PER semantics for (higher-order) observational equivalence

The abstract formulation of ASL+ studied in Chapter 5 introduced the idea of modelling specifications by relations on algebras rather than classes of algebras. This arose for the same reason that a types-as-sets semantics needs to be generalised to cope with the stratified equational theory of subtyping systems. As mentioned in Section 5.4.5, it would be interesting to see if a PER semantics for specifications is useful: integrating a relation of observational equivalence as part of the semantic denotation, rather than closing up the class of models under such an equivalence. There are connections with type systems for record subtyping, because of the obvious relationship between (dependent) record types and signatures in typical institutions (see the comments in Section 6.9.3). The new breed of type systems for modular programming pursues these ideas [Leroy, 1995, Harper and Lillibridge, 1994, Courant, 1997a].

Type theory like institutions

The motivation behind this new breed of type systems for modular programming is to come up with type-checking rules for module composition, which work at (or very close to) the level of the source language syntax. Re-examining ASL+$_{\mathrm{FPC}}$ with this idea in mind, the starting point of FPC signature in Definition 6.1 seems like an odd half-way house; probably FPC signatures and signature morphisms should be defined as in the abstract syntax of Section 6.8.2. Then the category $\mathbf{Sign}^{FPC}$ would be similar to a category of contexts as in the semantics of dependent types [Hofmann, 1997, Pitts, 1996]. Semantic signatures with a different order of declarations are isomorphic rather than identical.

A morphism between contexts in type theory semantics corresponds to a substitution; this is somewhat richer than renaming, and is like a *derived signature morphism* in the usual algebraic setting. Using a richer form of morphism between signatures allows a type constant in one signature to be mapped to a type expression over another signature. This means that sharing equations, which were added to signatures, could be removed again, and expressed with signature morphisms instead. But these are early thoughts, and derived signature morphisms are known to break some desirable algebraic properties.

This type-theoretic view is more finitary than the classical definitions for many-sorted algebras, but for the case of real programs and specifications, there seems to be nothing wrong in such limitations.

Type systems and proof search

An important research goal is to study mechanisms of proof search in structured specifications and programs. For the abstract version of ASL+ studied in Chapter 5, we want ways of reducing satisfaction judgements between higher-order objects to sets of proof obligations to be proved in the underlying proof systems. For the concrete version of ASL+ studied in Chapter 7, we want strategies for proof search in the satisfaction system. In both cases, there are parallels with the design of *algorithmic* versions of the type systems in question. But with specifications things seem more difficult: if we wish to prove a property φ of an application of a parameterised program $F(M)$, it is unrealistic to suppose that there is a "canonical" Π-specification $\Pi X{:}SP.\, SP'$ for F which always provides enough information to deduce φ in a *schematic* way. Some peculiar properties of M not assumed in SP may be required. In this case, we would need rules like the generalised λ-rule considered in Chapter 3, so that F can be given other Π-types which make use of particular properties of the actual argument M. We might want to combine such types together; this is an obvious application for intersection types (see also the example in Section 2.2.6).

An improved abstract ASL+

Another future task is to describe the construction of the module language of Chapter 7, or an improved version of it, in an abstract and institution-independent way. Some extra assumptions will be needed. The syntax and type-checking judgements of the core-language in Section 6.8 were described in a way suitable for abstraction, by relating pre-signatures to a class of signature inclusions, the signature extensions. (The definitions work just as well if we work with syntax for signatures throughout, as suggested above; inclusion between signatures can then be defined inductively over the length of the subsignature.) Moving to an abstract setting, we might postulate a special class of inclusions corresponding to extensions, and axiomatize the dot-notation renamings in terms of these. Related abstractions were introduced by Sannella and Tarlecki [1986], who defined the notion of *institution with syntax*. And there are several places already where the category **Sign** of an institution has been equipped with a system of inclusions, notably in Goguen [1991].

Mixing languages

Whilst the renaming approach of handling dot-notation described in Section 7.3.1 seems simple in outline, it turns out to be complicated in detail and has serious defects. I believe that a more satisfactory solution would be achieved by intimately combining the syntax and semantics of the module language with those of the core language, reflecting that the two language grammars are mutually-defined once we add modules and dot notation. Because the notion of core-level type-equality becomes dependent on the module-level context, this construction shows its true colours to be a simultaneous definition, rather than a construction which takes a core-level language and "adds modules on top", as some of the module-language folklore seems to suggest.

A similar thing happens when putting institutions together: to get proper "mixing" of the institutions, one has to use presentations of the syntax known as *parchments* [Goguen and Burstall, 1986]. The syntax of parchments can be combined to give the expected results [Tarlecki, 1996]. A similar point is made for the combination of logical encodings, described by Harper et al. [1994].

There might be ways of building ASL+ in an abstract way, following these ideas. In likelihood we would need to deal with contexts explicitly so recent ideas of *context institutions* and *context parchments* [Pawlowski, 1996] might be applicable. The parchment and institution theories do not apply directly, because they work with the sentences of institutions rather than syntax for signatures and models. But one can define institutions in which the sentences are programs, for example (see the comments in Section 5.10). Of course, once such an abstract construction was conceived, we would be obliged to persuade everyone that it does the right thing! So it would be worth revisiting the specific case of $\mathrm{ASL+}_{\mathrm{FPC}}$ defined in this manner.

8.3 Finally

I believe that the study in this thesis provides progress towards rigorous methods of programming and specification in-the-large. I have forged some new connections between parameterisation mechanisms used in specification languages and forms of parameterisation used in advanced type systems, and the work has lead to contributions in both areas. I have made preliminary investigations into new language designs, for solving some problems with existing languages.

Apart from specification and programming languages, advanced type systems with subtyping and module facilities have another important application: to automated proof assistants. One day we may see these proof assistants themselves form a basis for practical formal program development in a language with some features of ASL+. This is one of the ultimate aims of this research.

A *More about $\mathcal{FPC}$*

This appendix provides some more details about the formulation of the language FPC given in Chapter 6, and used as the core-language of ASL+$_{\text{FPC}}$ in Chapter 7. It also contains some details of a provisional satisfaction system for ASL+$_{\text{FPC}}$.

A.1 A Pervasive Environment for FPC

FPC is expressive enough to describe most of the common datatypes used in programming. Here we describe a collection of common types and operations on them, which might be part of a "pervasive environment" or standard library for a real programming language.

We take the view that the expressions written below are definitions in the meta-language. Some types and terms are parameterised by types, which is impossible in FPC itself since type functions cannot be expressed. We freely use subscripts, bracketed terms and "mixfix" notation on the left hand side to indicate where such parameters are resolved in the meta-language.

Empty and unit types

$$\begin{array}{lll} \mathit{void} & =_{\text{def}} & \mu a.a \\ \mathit{unit} & =_{\text{def}} & \mathit{void} \to \mathit{void} \\ \mathit{unity} & =_{\text{def}} & \textbf{fun}\,(x : \mathit{void}).\,x \end{array}$$

Boolean type

$$\begin{aligned}
&\mathit{bool} && =_{\mathrm{def}} && \mathit{unit} + \mathit{unit}\\
&\mathbf{true} && =_{\mathrm{def}} && \mathbf{inl}_{\mathit{bool}}(\mathit{unity})\\
&\mathbf{false} && =_{\mathrm{def}} && \mathbf{inr}_{\mathit{bool}}(\mathit{unity})\\
&\mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 && =_{\mathrm{def}} && \mathbf{case}\ b\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2
\end{aligned}$$

Option type

$$\begin{aligned}
&\mathit{option}_t && =_{\mathrm{def}} && t + \mathit{unit}\\
&\mathit{some}_t(e) && =_{\mathrm{def}} && \mathbf{inl}_{\mathit{option}_t}(e)\\
&\mathit{none}_t && =_{\mathrm{def}} && \mathbf{inr}_{\mathit{option}_t}(\mathit{unity})
\end{aligned}$$

Fixed point operator

$$\begin{aligned}
\mathit{FIX}_{s,t} \quad =_{\mathrm{def}} \quad & \mathbf{fun}\,(f : (s \to t) \to (s \to t)).\\
& \quad (\mathbf{fun}\,(x : r).\ \mathbf{fun}\,(y : s).\ f\ (\mathbf{elim}(x)\ x)\ y)\\
& \quad (\mathbf{intro}_r(\mathbf{fun}\,(x : r).\ \mathbf{fun}\,(y : s).\ f\ (\mathbf{elim}(x)\ x)\ y))
\end{aligned}$$

where $r \equiv \mu a.a \to (s \to t)$

$$\perp_s \quad =_{\mathrm{def}} \quad \mathit{FIXs}, s\ (\mathbf{fun}\,(f : s \to s).\, f)$$

$$\mathit{funrec}\ f\ (x : s) : t.\, e \quad =_{\mathrm{def}} \quad \mathit{FIX}_{s,t}(\mathbf{fun}\,(f : s \to t).\ \mathbf{fun}\,(x : s).\, e)$$

Natural numbers type

$$\begin{aligned}
&\mathit{nat} && =_{\mathrm{def}} && \mu a.\mathit{unit} + a\\
&0 && =_{\mathrm{def}} && \mathbf{intro}_{\mathit{nat}}(\mathbf{inl}_{\mathit{unit}+\mathit{nat}}(\mathit{unity}))\\
&\mathit{succ}(e) && =_{\mathrm{def}} && \mathbf{intro}_{\mathit{nat}}(\mathbf{inr}_{\mathit{unit}+\mathit{nat}}(e))\\
&\mathit{pred}(e) && =_{\mathrm{def}} && \mathbf{case}\ \mathbf{elim}(e)\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow 0\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow y\\
&\mathit{zero?}(e) && =_{\mathrm{def}} && \mathbf{case}\ \mathbf{elim}(e)\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow \mathbf{true}\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow \mathbf{false}\\
&\mathit{iter}_t && =_{\mathrm{def}} && \mathbf{fun}\,(z : t).\\
& && && \quad \mathbf{fun}\,(s : t \to t).\\
& && && \qquad \mathit{funrec}\ \mathit{iter}'\ (n : \mathit{nat}) : t.\\
& && && \qquad\quad \mathbf{if}\ \mathit{zero?}(n)\ \mathbf{then}\ z\ \mathbf{else}\ s\ (\mathit{iter}'\ (\mathit{pred}(n)))
\end{aligned}$$

Clearly we can define other familiar functions on the datatypes, as well as other datatypes such as list_t.

A.2 Operational Semantics of FPC

The operational semantics of FPC expressions are shown in Figure A.1 on the next page. The rules axiomatise an evaluation relation on expressions e, independent of typing. If $e \Downarrow r$ we say that e *evaluates* to value r. The *values*, r, are the subset of the closed expressions e definable in the grammar:

$$\begin{array}{rcl} r & ::= & v \mid x \mid \mathbf{fun}\,(x:t).\,e \mid \langle r, r\rangle \\ & \mid & \mathbf{inl}_{t_1+t_2}(r) \mid \mathbf{inr}_{t_1+t_2}(r) \mid \mathbf{intro}_{\mu a.t}(r) \end{array}$$

The rule for function application makes this a call-by-value operational semantics (and the denotational semantics given in Chapter 6 corresponds with this).

The following proposition summarizes two standard properties of the operational semantics of FPC.

Proposition A.1 (Operational semantics of FPC).

1. *(Closure of typing under evaluation relation).*

   If $G \rhd^{\Sigma} e : t$ and $e \Downarrow r$, then $G \rhd^{\Sigma} r : t$ also.

2. *(Soundness and Adequacy wrt denotational semantics in Definition 6.13 on page 201).*

   (a) If $e \Downarrow r$ then $\mathcal{M}[\![e]\!]_{\iota\rho} = \mathcal{M}[\![r]\!]_{\iota\rho}$; if $e \not\Downarrow$ then $\mathcal{M}[\![e]\!]_{\iota\rho} = \bot$

   (b) If $\mathcal{M}[\![e]\!]_{\iota\rho} = \mathcal{M}[\![e']\!]_{\iota\rho}$ and $e \Downarrow r$, then $e' \Downarrow r$ also.

   for suitable environments ι, ρ.

Proof See Gunter [1992]. □

A.3 Derived Connectives of LFPC

Higher-order logic extended with FPC was defined in Section 6.4. Only a few primitive connectives are needed and then the truth values, conjunction, disjunction and quantifiers can all be defined. Here are suitable definitions using the primitives of LFPC:

$$\frac{}{v \Downarrow v}$$

$$\frac{}{x \Downarrow x}$$

$$\frac{}{\mathbf{fun}\,(x:t).\,e \Downarrow \mathbf{fun}\,(x:t).\,e}$$

$$\frac{e_1 \Downarrow \mathbf{fun}\,(x:t).\,e' \qquad e_2 \Downarrow r' \qquad e'[r'/x] \Downarrow r}{e_1\,e_2 \Downarrow r}$$

$$\frac{e_1 \Downarrow r_1 \qquad e_2 \Downarrow r_2}{\langle e_1, e_2\rangle \Downarrow \langle r_1, r_2\rangle}$$

$$\frac{e \Downarrow \langle r_1, r_2\rangle}{\mathbf{fst}(e) \Downarrow r_1}$$

$$\frac{e \Downarrow \langle r_1, r_2\rangle}{\mathbf{snd}(e) \Downarrow r_2}$$

$$\frac{e_1 \Downarrow r_1}{\mathbf{inl}_{t_1+t_2}(e_1) \Downarrow \mathbf{inl}_{t_1+t_2}(r_1)}$$

$$\frac{e_2 \Downarrow r_2}{\mathbf{inr}_{t_1+t_2}(e_2) \Downarrow \mathbf{inr}_{t_1+t_2}(r_2)}$$

$$\frac{e \Downarrow \mathbf{inl}(r_1) \qquad e_1[r_1/x] \Downarrow r}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2 \Downarrow r}$$

$$\frac{e \Downarrow \mathbf{inr}(r_2) \qquad e_2[r_2/y] \Downarrow r}{\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_1\ \mathbf{or}\ \mathbf{inr}(y) \Rightarrow e_2 \Downarrow r}$$

$$\frac{e \Downarrow r}{\mathbf{intro}_{\mu a.t}(e) \Downarrow \mathbf{intro}_{\mu a.t}(r)}$$

$$\frac{e \Downarrow \mathbf{intro}_{\mu a.t}(r)}{\mathbf{elim}_{\mu a.t}(e) \Downarrow r}$$

Figure A.1: Call-by-value operational semantics of FPC

$$\begin{array}{lll}
\mathsf{T} & =_{\text{def}} & (\Lambda p{:}\mathbf{prop}.\, p) \;=\; (\Lambda p{:}\mathbf{prop}.\, p) \\
\forall_\tau & =_{\text{def}} & \Lambda P{:}\tau \to \mathbf{prop}.\, P = \Lambda z{:}\tau.\, \mathsf{T} \\
\exists_\tau & =_{\text{def}} & \Lambda P{:}\tau \to \mathbf{prop}.\, P\;\; (\epsilon z{:}\tau.\, P\; z) \\
\mathsf{F} & =_{\text{def}} & \forall p : \mathbf{prop}.\, p \\
\neg & =_{\text{def}} & \Lambda p{:}\mathbf{prop}.\, p \Rightarrow \mathsf{F} \\
\bigwedge & =_{\text{def}} & \Lambda p_1, p_2{:}\mathbf{prop}.\, \forall p : \mathbf{prop}.\, (p_1 \Rightarrow (p_2 \Rightarrow p)) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \Rightarrow p \\
\bigvee & =_{\text{def}} & \Lambda p_1, p_2{:}\mathbf{prop}.\, \forall p : \mathbf{prop}.\, (p_1 \Rightarrow p) \Rightarrow \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (p_2 \Rightarrow p) \Rightarrow p \\
h_1 \neq h_2 & =_{\text{def}} & \neg\, (h_1 = h_2)
\end{array}$$

We have used $\forall z : \tau.\, P$ in place of $\forall_\tau\; (\Lambda z{:}\tau.\, P)$. We also write $\forall y, z : \tau.\, P$ to abbreviate $\forall y : \tau.\, \forall z : \tau.\, P$, and so on for more than two variables. Similarly for $\exists$. We also use the usual infix notation for $\bigwedge$ and $\bigvee$.

The familiar rules for these connectives are derivable from those given in the next section.

A.4 A Proof System for LFPC

Here we present a bare proof system for LFPC, consisting of structural rules for the primitive connectives of the logic and axioms for the $\sqsubseteq$ operator. Some axioms for $\sqsubseteq$ were already mentioned in Section 6.6 on page 207.

The rules can be formulated in a multitude of ways (see for example Gordon and Melham [1993], Lambek and Scott [1986]); here we give a simple formulation which avoids dependency between the primitive connectives. This results in some redundancy, for example, the cut rule may be derived from the implication rules.

Recall that the proof system derives sequents with the form:

$$\Phi \vdash^{\Sigma} \varphi \;\; [G]$$

where Σ is an FPC signature and G is a LFPC context, describing the constants and free variables of the formula φ and the set of formulae Φ. The context $[G]$ is often omitted for brevity; in most rules, the context is fixed.

Structural axioms and rules

$$\frac{G \rhd^{\Sigma} \varphi : \mathbf{prop}}{\varphi \vdash^{\Sigma} \varphi \;\; [G]}$$

$$\frac{\Phi \vdash^{\Sigma} \varphi \ [G] \qquad G \rhd^{\Sigma} \varphi' : \mathbf{prop}}{\Phi, \varphi' \vdash^{\Sigma} \varphi \ [G]}$$

$$\frac{\Phi \vdash^{\Sigma} \varphi \qquad \Phi, \varphi \vdash^{\Sigma} \varphi'}{\Phi \vdash^{\Sigma} \varphi'}$$

$$\frac{\Phi \vdash^{\Sigma} \varphi \qquad G \subseteq G'}{\Phi \vdash^{\Sigma} \varphi \ [G']}$$

$$\frac{G \rhd^{\Sigma} h : \tau \qquad \Phi \vdash^{\Sigma} \varphi \ [G, z : \tau, G']}{\Phi[h/z] \vdash^{\Sigma} \varphi[h/z] \ [G, G']}$$

Equality axioms and rules

$$\frac{G \rhd^{\Sigma} h : \tau}{\vdash^{\Sigma} h = h \ [G]}$$

$$\frac{G \rhd^{\Sigma} h_1, h_2 : \tau \qquad G, z : \tau \rhd^{\Sigma} \varphi : \mathbf{prop}}{h_1 = h_2, \ \varphi[h_1/z] \vdash^{\Sigma} \varphi[h_2/z] \ [G]}$$

$$\frac{G \rhd^{\Sigma} \varphi_1, \varphi_2 : \mathbf{prop} \qquad \Phi, \varphi_1 \vdash^{\Sigma} \varphi_2 \ [G] \qquad \Phi, \varphi_2 \vdash^{\Sigma} \varphi_1 \ [G]}{\Phi \vdash^{\Sigma} \varphi_1 = \varphi_2}$$

$$\frac{G, z : \tau \rhd^{\Sigma} h : \tau' \qquad G \rhd^{\Sigma} h' : \tau}{\vdash^{\Sigma} (\Lambda z{:}\tau.\, h)\, h' \ = \ h[h'/z] \ [G]}$$

$$\frac{G \rhd^{\Sigma} f : \tau \to \tau'}{\vdash^{\Sigma} \Lambda z{:}\tau.\, f\, z \ = \ f \ [G]}$$

$$\frac{G, z : \tau \rhd^{\Sigma} h_1, h_2 : \tau'}{h_1 = h_2 \vdash^{\Sigma} \Lambda z{:}\tau.\, h_1 \ = \ \Lambda z{:}\tau.\, h_2 \ [G, z : \tau]}$$

$$\vdash^{\Sigma} \forall x : \mathbf{prop}.\, x = \mathsf{T} \ \bigvee \ x = \mathsf{F}$$

Implication rules

$$\frac{\Phi, \varphi_1 \vdash^{\Sigma} \varphi_2}{\Phi \vdash^{\Sigma} \varphi_1 \Rightarrow \varphi_2} \qquad \frac{\Phi \vdash^{\Sigma} \varphi_1 \Rightarrow \varphi_2 \qquad \Phi \vdash^{\Sigma} \varphi_1}{\Phi \vdash^{\Sigma} \varphi_2}$$

Choice axiom

$$\frac{G, z : \tau \rhd^{\Sigma} \varphi : \mathbf{prop}}{\varphi \vdash^{\Sigma} \varphi[\epsilon z{:}\tau.\, \varphi/z] \;\; [G, z : \tau]}$$

Order relation

$$\vdash^{\Sigma} \forall x : t.\, x \sqsubseteq x$$

$$\vdash^{\Sigma} \forall x, y, z : t.\, x \sqsubseteq y \;\wedge\; y \sqsubseteq z \Rightarrow x \sqsubseteq z$$

$$\vdash^{\Sigma} \forall x, y : t.\, x \sqsubseteq y \;\wedge\; y \sqsubseteq x \Rightarrow x = y$$

$$\vdash^{\Sigma} \forall f : s \rightharpoonup t.\, \forall x, y : s.\, x \sqsubseteq y \Rightarrow \; f\, x \sqsubseteq f\, y$$

$$\vdash^{\Sigma} \forall x : t.\, \bot_t \sqsubseteq x$$

$$\vdash^{\Sigma} \forall f : (s \rightharpoonup t) \rightharpoonup (s \rightharpoonup t).\, \mathit{FIX}_{s,t}\, f \;=\; f(\mathit{FIX}_{s,t}\, f)$$

FPC types are CPOs

Sets in HOL Sets in HOL are represented as predicates; rather than a single type of sets, we have $set_\tau =_{\text{def}} \tau \to \mathbf{prop}$. Much useful derived notation can be defined, including:

$$\begin{array}{lll}
x \in S & =_{\text{def}} & S(x) \\
\forall x \in S.\, P(x) & =_{\text{def}} & \forall x : \tau.\, S(x) \Rightarrow P(x) \\
\exists x \in S.\, P(x) & =_{\text{def}} & \exists x : \tau.\, S(x) \;\wedge\; P(x) \\
S \to S' & =_{\text{def}} & \Lambda f{:}\tau \to \tau'.\, \forall x \in S.\, f(x) \in T \\
\mathsf{pow}(S) & =_{\text{def}} & \Lambda T{:}\tau \to \mathbf{prop}.\, \forall x : \tau.\, x \in T \Rightarrow x \in S
\end{array}$$

By convention, $S : set_\tau$. Some type information will be omitted for clarity. The following definitions are a direct translation of the standard definitions from Section 6.3.

$$\begin{array}{lll}
\mathtt{UB}(j,S) & =_{\text{def}} & \forall x \in S.\, x \sqsubseteq j \\
\mathtt{LUB}(j,S) & =_{\text{def}} & \forall x \in S.\, \mathtt{UB}(x,S) \Rightarrow j \sqsubseteq x \\
\bigsqcup S & =_{\text{def}} & \epsilon j{:}\tau.\, \mathtt{LUB}(j,S) \\
\mathtt{Injective}(f,S) & =_{\text{def}} & \forall x \in S.\, \forall y \in S.\, x \neq y \Rightarrow f(x) \neq f(y) \\
\mathtt{Onto}(f,S,S') & =_{\text{def}} & \forall y \in S'.\, \exists x \in S.\, f(x) = y \\
\mathtt{Finite}(S) & =_{\text{def}} & \forall f \in S \to S.\, \mathtt{Injective}(f) \Rightarrow \mathtt{Onto}(f) \\
\mathtt{Directed}(S) & =_{\text{def}} & \\
 & & \forall T \in \mathit{pow}(S).\, \mathtt{Finite}(T) \Rightarrow \exists j \in S.\, \mathtt{UB}(j,S) \\
\mathtt{Complete}(S) & =_{\text{def}} & \\
 & & \forall T \in \mathit{pow}(S).\, \mathtt{Directed}(S) \Rightarrow \exists j \in S.\, \mathtt{LUB}(j,S)
\end{array}$$

Here is the single axiom which characterises FPC types as CPOs. The term $\Lambda x{:}t.\,\mathsf{T}$ represents the underlying set of the domain which interprets t.

$$\vdash^{\Sigma} \mathtt{Complete}(\Lambda x{:}t.\,\mathsf{T})$$

Fixed Point Induction

Suppose P is a predicate over some FPC type, $P : t \to \mathbf{prop}$.

$$\begin{array}{l}
\mathtt{Admissible}(P) \quad =_{\text{def}} \\
\quad \forall S \in \mathit{pow}(\Lambda x{:}t.\,\mathsf{T}).\ \mathtt{Directed}(S) \\
\qquad\qquad\qquad \Rightarrow (\forall x \in S.\, P(x)) \Rightarrow P(\bigsqcup S)
\end{array}$$

The rule of fixed point induction is:

$$\frac{\Phi \vdash^{\Sigma} \mathtt{Admissible}(P) \qquad \begin{array}{l}\Phi \vdash^{\Sigma} P(\bot_t) \\ \Phi \vdash^{\Sigma} \forall x : t.\, P(x) \Rightarrow P(f x)\end{array}}{\Phi \vdash^{\Sigma} P(\mathit{FIX}\, f)}$$

Axioms for functions

$$\frac{G \rhd^{\Sigma} e : s}{\vdash^{\Sigma} \bot_{s \rightarrowtail t}\, e \;=\; \bot_t \;\; [G]}$$

$$\frac{G \rhd^{\Sigma} e : s \rightarrowtail t}{\vdash^{\Sigma} e\, \bot_s \;=\; \bot_t \;\; [G]}$$

$$\frac{G, x : s \rhd^{\Sigma} e : t \qquad G \rhd^{\Sigma} e' : s}{\vdash^{\Sigma} e' \neq \bot_s \Rightarrow (\mathbf{fun}\,(x : s).\, e)\; e' \;=\; (\Lambda x{:}s.\, e) e' \;\; [G]}$$

Axioms for products and sums

The axioms characterising the product and sum constructors correspond directly to the definition of the semantics in Definition 6.13. As usual with LCF, there is a proliferation of axioms concerning definedness and strictness, as well as the uniqueness and "evaluation" axioms.

In the axioms below, it is assumed that $G \rhd^{\Sigma} e_1 : s$, $G \rhd^{\Sigma} e_2 : t$ and $G \rhd^{\Sigma} p : s \times t$. All the axioms also have context G.

$$\vdash^{\Sigma} \langle \mathbf{fst}(p), \mathbf{snd}(p) \rangle = p$$

$$\vdash^{\Sigma} e_1 \neq \bot_s \bigwedge e_2 \neq \bot_t \Longrightarrow \langle e_1, e_2 \rangle \neq \bot_{s\times t}$$

$$\vdash^{\Sigma} e_1 = \bot_s \bigvee e_2 = \bot_t \Longrightarrow \langle e_1, e_2 \rangle = \bot_{s\times t}$$

$$\vdash^{\Sigma} \mathbf{fst}(\bot_{s\times t}) = \bot_s \qquad \vdash^{\Sigma} \mathbf{fst}(\langle e_1, e_2 \rangle) = e_1$$

$$\vdash^{\Sigma} \mathbf{snd}(\bot_{s\times t}) = \bot_t \qquad \vdash^{\Sigma} \mathbf{snd}(\langle e_1, e_2 \rangle) = e_2$$

In the axioms below, it also assumed that $G, x : s \rhd^{\Sigma} f_1 : u$ and $G, y : t \rhd^{\Sigma} f_2 : u$.

$$\vdash^{\Sigma} \mathbf{inl}_{s+t}(e_1) \neq \mathbf{inr}_{s+t}(e_2)$$

$$\vdash^{\Sigma} e_1 \neq \bot_s \Longrightarrow \mathbf{inl}_{s+t}(e_1) \neq \bot_{s+t}$$

$$\vdash^{\Sigma} e_2 \neq \bot_t \Longrightarrow \mathbf{inl}_{s+t}(e_2) \neq \bot_{s+t}$$

$$\vdash^{\Sigma} e_1 = \bot_s \Longrightarrow \mathbf{inl}_{s+t}(e_1) = \bot_{s+t}$$

$$\vdash^{\Sigma} e_2 = \bot_t \Longrightarrow \mathbf{inl}_{s+t}(e_2) = \bot_{s+t}$$

$$\vdash^{\Sigma} e \neq \bot_{s+t} \Longrightarrow (\mathbf{case}\ e\ \mathbf{of}\, \mathbf{inl}(x) \Rightarrow f_1\ \mathbf{or}\, \mathbf{inr}(y) \Rightarrow f_2) \neq \bot_u$$

$$\vdash^{\Sigma} \mathbf{case}\ \bot_{s+t}\ \mathbf{of}\, \mathbf{inl}(x) \Rightarrow f_1\ \mathbf{or}\, \mathbf{inr}(y) \Rightarrow f_2 = \bot_u$$

$$\vdash^{\Sigma} \mathbf{case}\ \mathbf{inl}_{s+t}(e_1)\ \mathbf{of}\, \mathbf{inl}(x) \Rightarrow f_1\ \mathbf{or}\, \mathbf{inr}(y) \Rightarrow f_2 = f_1[e_1/x]$$

$$\vdash^{\Sigma} \mathbf{case}\ \mathbf{inr}_{s+t}(e_2)\ \mathbf{of}\, \mathbf{inl}(x) \Rightarrow f_1\ \mathbf{or}\, \mathbf{inr}(y) \Rightarrow f_2 = f_2[e_2/y]$$

Axioms for recursive types

The term constructors are characterised as isomorphisms.

$$\vdash^{\Sigma} \mathbf{intro}_{\mu a.t}(\bot_{[\mu a.t/a]t}) = \bot_{\mu a.t}$$

$$\frac{G \rhd^{\Sigma} e : [\mu a.t/a]t}{\vdash^{\Sigma} \mathbf{elim}(\mathbf{intro}_{\mu a.t}(e)) = e}$$

$$\frac{G \rhd^{\Sigma} e : \mu a.t}{\vdash^{\Sigma} \mathbf{intro}_{\mu a.t}(\mathbf{elim}(e)) = e}$$

Bibliography

S. Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, BRICS, Department of Computer Science, University of Aarhus, 1994a.

S. Agerholm. LCF examples in HOL. *The Computer Journal*, 38(2), 1994b.

D. Aspinall. Subtyping with singleton types. In *Proc. Computer Science Logic, CSL'94, Kazimierz, Poland*, Lecture Notes in Computer Science 933. Springer-Verlag, 1995a.

D. Aspinall. Types, subtypes, and ASL+. In *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 906. Springer-Verlag, 1995b.

D. Aspinall and A. 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.

A. Avron. Simple consequence relations. *Information and Computation*, 92: 105–139, 1991.

A. Avron. Axiomatic systems, deduction and implication. *J. Logic Computation*, 2(1):51–98, 1992.

A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

A. Bailey. LEGO with implicit coercions (draft). Technical report, Department of Computer Science, University of Manchester, April 1996.

H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

J. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37 (2):335–372, April 1990.

G. Betarte and A. Tasistro. Formalisation of systems of algebras using dependent record types and subtyping: an example. In *Proceedings of the 7th Nordic Workshop on Programming Theory, Report 86 PMG, Göteborg, Sweden*, 1996.

G. Betarte and A. Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In *Proceedings of 25 Years of Constructive Type Theory*. Oxford University Press, 1997.

V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stölen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Institut für Informatik, Technische Universität München, May 1993a.

M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stölen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Institut für Informatik, Technische Universität München, May 1993b.

K. Bruce and J. 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.

K. B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87:196–240, 1990.

R. M. Burstall. Programming with modules as typed functional programming. In *Proc. International Conference on Fifth Generation Computing Systems, ICOT, Tokyo*, 1984.

R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specificiation language. In *Proceedings of Advanced Course on Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 292–232. Springer-Verlag, 1980.

R. M. Burstall and J. H. McKinna. Deliverables: an approach to program development in Constructions. Technical Report ECS-LFCS-91-133, LFCS, Department of Computer Science, University of Edinburgh, 1991.

L. 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.

L. Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, Oct. 1990.

L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software, Sendai, Japan, September 1991*, Lecture Notes in Computer Science 526. Springer-Verlag, 1992.

M. V. Cengarle. *Formal Specifications with Higher-Order Parameterisation.* PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 1994.

M. V. Cengarle and M. Wirsing. A calculus of parameterization for algebraic specifications. Technical Report 94/198, Department of Computer Science, Monsahs University, 1994.

G. Chen. Subtyping calculus of constructions. In *Proceedings of 22nd International Symposium, MFCS '97*, volume Lecture Notes in Computer Science 1295, pages 189–198. Springer-Verlag, 1997.

G. Chen and G. Longo. Subtyping parametric and dependent types. Technical report, Département de Mathématiques et d'Informatique, Ecole Normale Supérieure, Paris, December 1996.

CoFI. The Common Framework Initiative for Algebraic Specification and Development — Rationale. Available by FTP or on the WWW as `http://www.brics.dk/Projects/CoFI/Documents/Rationale/`, May 1997.

CoFI Task Group on Language Design. CASL — The CoFI Algebraic Specification Language — Rationale. Available by FTP or on the WWW as `http://www.brics.dk/Projects/CoFI/Documents/CASL/Rationale/`, May 1997.

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.

A. B. Compagnoni. *Higher-Order Subtyping with Intersection Types.* PhD thesis, Nijmegen Catholic University, 1995.

A. B. Compagnoni and B. C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996.

J. Courant. An applicative module calculus. In *Proceedings TAPSOFT '97*, pages 622-636. Springer-Verlag LNCS 1214, October 1997a.

J. Courant. A module calculus for Pure Type Systems. In P. Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications TLCA '97*. Springer-Verlag LNCS 1210, 1997b.

P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

L. Dami. Functions, records and compatibility in the Lambda N calculus. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 153–174. Prentice Hall, 1995.

N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal Models and Semantics – Handbook of Theoretical Computer Science*, volume B, chapter 6. Elsevier, 1990.

J. Farrés-Casals. *Verification in ASL and related Specification Languages*. PhD thesis, Edinburgh University, 1992.

L. Feijs. The calculus $\lambda\pi$. In M. Wirsing and J. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Computer Science 394, pages 307–328. Springer-Verlag, 1989.

J. S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In *Proceedings VDM'90 Conference, Kiel*, Lecture Notes in Computer Science 428, pages 189–210. Springer-Verlag, 1990.

P. Gardner. *Representing Logics in Type Theory*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

H. Geuvers. *Logics and Type Systems*. PhD thesis, Nijmegen Catholic University, 1993.

H. Geuvers and M.-J. Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2): 155–189, Apr. 1991.

H. Geuvers and B. 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.

G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.

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

J. A. Goguen. Types as theories. In M. Reed, A. W. Roscoe, and R. F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.

J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In D. H. Pitts, editor, *Proc. Workshop on Category Theory and Computer Programming*, Lecture Notes in Computer Science 240. Springer-Verlag, 1986.

J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.

M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

T. G. Griffin. Notational definition—a formal account. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 372–383, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

R. Grosu and D. Nazareth. Towards a New Way of Parameterization. In *Proceedings of the Third Maghrebian Conference on Software Engineering and Artificial Intelligence*, pages 383–392, 1994.

C. A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* MIT Press, 1994.

C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Formal Models and Semantics - Handbook of Theoretical Computer Science*, volume B, chapter 12. Elsevier, 1990.

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

R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 123–137, Portland, Oregon, January 17–21, 1994. ACM Press.

R. Harper and R. Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

R. Harper, D. Sannella, and A. Tarlecki. Logic representation in LF. In *Proceedings of the 3rd Summer Conference on Category Theory and Computer Science*, pages 250–272. Springer-Verlag, 1989.

R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

S. Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109, 1994.

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

M. Hofmann. Syntax and semantics of dependent types. In P. Dybjer and A. Pitts, editors, *Semantics of Logics of Computation*, chapter 3. Cambridge University Press, 1997.

M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science*, 167:3–45, 1996.

B. Jacobs and T. 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.

H. B. M. Jonkers. An introduction to COLD-K. In M. Wirsing and J. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Computer Science 394, pages 139–205. Springer-Verlag, 1989.

S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, LFCS, Department of Computer Science, University of Edinburgh, August 1994.

B. Krieg-Brückner and D. Sannella. Structuring specifications in-the-large and in-the-small: Higher-order functions, dependent types and inheritance in SPECTRAL. In *Proc. Colloq. on Combining Paradigms for Software Development. Intl. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91)*, Lecture Notes in Computer Science 494, pages 103–120. Springer, 1991.

J. Lambek and P. J. Scott. *Introduction to higher order categorical logic.* Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986. First paperback edition (with corrections) 1988.

B. Lampson and R. Burstall. A kernel language for abstract data types and modules. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, Lecture Notes in Computer Science 173, pages 1–50, Sophia-Antipolis, France, June 1984. Springer-Verlag.

B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76:278–346, February/March 1988. An earlier version appeared as [Lampson and Burstall, 1984].

X. Leroy. Manifest types, modules, and separate compilation. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 109–122, Portland, Oregon, January 17–21, 1994. ACM Press.

X. Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 142–153, San Francisco, California, January 22–25, 1995. ACM Press.

X. Leroy. A modular module system. Research report 2866, INRIA, Apr. 1996a.

X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996b.

J. Levy, J. Agustí, F. Esteva, and P. García. An ideal model for an extended λ-calculus with refinement. LFCS ECS-LFCS-91-188, Edinburgh University, November 1991.

G. Longo, K. Milsted, and S. 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.

Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3(3), 1993.

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.

Z. Luo. Coercive subtyping in type theory. In *Proc. Computer Science Logic, CSL'96, Utrecht*, 1996.

D. MacQueen. Using dependent types to express modular structure. In *Proceedings, Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT-SIGPLAN, ACM Press.

D. MacQueen. A higher-order type system for functional programming. In D. A. Turner, editor, *Research Topics in Functional Programming*, chapter 13, pages 353–367. Addison-Wesley, 1991.

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

J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium '87*, pages 275–329. Elsevier, 1989.

A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

P. D. Mosses. Unified algebras and modules. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 329–343, Austin, Texas, January 11–13, 1989. ACM SIGACT-SIGPLAN, ACM Press.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction.* Oxford University Press, 1990.

P. Odifreddi, editor. *Logic and Computer Science.* Academic Press, 1990.

D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

L. C. Paulson. *Logic and Computation.* Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

L. C. Paulson. *Isabelle — A Generic Theorem Prover.* Lecture Notes in Computer Science 828. Springer-Verlag, 1994.

W. Pawlowski. Context institutions. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types joint with the 8th general COMPASS workshop*, pages 436–457. Springer-Verlag, 1996.

F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 315-328, May 1993.

B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

B. C. Pierce. Bounded quantification is undecidable. In *Proceedings, Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 305–315, Albuquerque, New Mexico, January 19–22, 1992. ACM Press.

B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1), July 1994.

B. C. Pierce and R. Pollack. Higher-order subtyping. Draft notes. Department of Computer Science, University of Edinburgh, 1992.

B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994.

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

G. Plotkin. Denotational semantics with partial functions. Lecture at C.S.L.I. Summer School, 1985.

E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven Technical University, 1994.

F. Regensburger. *HOLCF: A conservative extension of HOL with LCF.* PhD thesis, Technische Universität München, 1994.

B. Reus and T. Streicher. Lifting the laws of module algebra to deliverables. Technical Report 9203, Ludwig-Maximilians-Universitt, München, August 1992.

C. V. Russo. *Types for program modules.* PhD thesis, University of Edinburgh, 1997. Forthcoming.

D. Sannella and R. M. Burstall. Structured theories in LCF. In *Proceedings of the 8th Colloquium on Trees in Algebra and Programming*, Mar. 1983.

D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. Technical Report 6/90, FB Informatik, Universität Bremen, April 1990.

D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

D. Sannella and A. Tarlecki. Extended ML: An institution-independent framework for formal program development. In D. H. Pitts, editor, *Proc. Workshop on Category Theory and Computer Programming*, Lecture Notes in Computer Science 240, pages 364–389. Springer-Verlag, 1986.

D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, 1988a.

D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica*, 25: 233–281, 1988b.

D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS–LFCS–89–71, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989. Full version of abstract that appeared in Proc. TAPSOFT '89, Barcelona. Springer LNCS 352, 375-389.

D. Sannella and A. 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.

D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Model-theoretic foundations. In *19th International Colloquium on Automata, Languages and Programming*, volume 623, pages 656–671. Springer LNCS, 1992.

D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 1997.

D. Sannella and L. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.

D. Sannella and M. Wirsing. Implementation of parameterised specifications. In *Proc. 9th ICALP, Aarhus*, volume Lecture Notes in Computer Science 140, pages 473–488. Springer-Verlag, 1982.

D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proceedings of International Conference on Foundations of Computation Theory, Borgholm, Sweden*, Lecture Notes in Computer Science 158. Springer-Verlag, 1983.

A. Scedrov. A guide to polymorphic types. In Odifreddi [1990].

O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, Department of Computer Science, The University of Edinburgh, February 1987.

P. Severi and E. Poll. Pure Type Systems with Definitions. In *Logical Foundations of Computer Science, LFCS'94*, Lecture Notes in Computer Science 813, pages 316–328. Springer-Verlag, 1994.

S. Sokołowski. Parametricity in algebraic specifications: A case study. Technical report, Institute of Computer Science, Polish Academy of Sciences, Gdańsk, 1989.

M. Spivey. Richer types for Z. *Formal Aspects of Computing*, 8:565–584, 1996.

M. Steffen and B. Pierce. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1994.

T. Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991.

T. Streicher and M. Wirsing. Dependent types considered necessary for specification languages. In H. Ehrig et al., editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 534. Springer-Verlag, 1991.

A. Tarlecki. Modules for a model-oriented specification language: a proposal for MetaSoft. In *Proc. 4th European Symposium on Programming ESOP'92*, Lecture Notes in Computer Science 582, pages 452–472. Springer-Verlag, 1992.

A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types joint with the 8th general COMPASS workshop*, pages 478–502. Springer-Verlag, 1996.

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

M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42:123–249, 1986.

M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Formal Models and Semantics – Handbook of Theoretical Computer Science*, volume B, chapter 13. Elsevier, 1990.

M. Wirsing. Proofs in structured specifications. Technical Report MIP-9008, Universität Passau, 1991.

Index