The Formal Semantics of PVS[1]

Sam Owre and Natarajan Shankar
`owre@csl.sri.com` `shankar@csl.sri.com`
URL: `http://www.csl.sri.com/sri-csl-fm.html`
SRI International
Computer Science Laboratory
Menlo Park CA 94025 USA

Technical Report CSL-97-2R
August 1997, Revised March 1999

[1]Funded by National Aeronautics and Space Administration Contract NAS1-18969, Task 11 and National Science Foundation Grant CCR 9300444.

Abstract

A specification language is a medium for expressing *what* is computed rather than *how* it is computed. Specification languages share some features with programming languages but are also different in several important ways. For our purpose, a specification language is a logic within which the behavior of computational systems can be formalized. Although a specification can be used to simulate the behavior of such systems, we mainly use specifications to state and prove system properties with mechanical assistance.

We present the formal semantics of the specification language of SRI's Prototype Verification System (PVS). This specification language is based on the simply typed lambda calculus. The novelty in PVS is that it contains very expressive language features whose static analysis (e.g., typechecking) requires the assistance of a theorem prover. The formal semantics illuminates several of the design considerations underlying PVS, particularly the interaction between theorem proving and typechecking.

Contents

Chapter 1

Introduction

PVS is a system for specifying and verifying properties of digital hardware and software systems. The specification language of PVS is designed to admit succinct, readable, and logically meaningful specifications. The PVS specification language is designed for effective proof construction rather than efficient execution. The design considerations underlying the language are therefore somewhat different from those of a corresponding programming language. For example, the language contains constructs that can be statically typechecked only with the assistance of a theorem prover. This is acceptable because the PVS specification language is intended for use in conjunction with powerful support for automated theorem proving. The logic of PVS is based on a simply typed higher-order logic with function, record, and product types, and recursive type definitions. This type system is extended with subtypes that are analogous to subsets, and with dependently typed functions, records, and products. The resulting type system has several advantages. It is possible, for instance, to statically ensure that all array references are within their respective array bounds. PVS specifications are organized into theories that can be parametric in types as well as individuals. While the semantics of the simply typed fragment is straightforward, the extensions such as subtyping, dependent typing, and (theory-level) parametricity do pose significant challenges. This report presents a concise but idealized definition of the PVS specification language and its intended formal set-theoretic semantics. It is neither an overview of the PVS language nor a guide to the Prototype Verification System (see the PVS user manuals [OSRSC98]).

The primary purpose of the formal semantics is as a useful reference for the developers and users of PVS. The idealized core of the specification language as presented here serves as a succinct foundation for studying the expressive power of the language. Pertinent questions about PVS are answered directly by the formal semantics presented here:

1. What is the semantic core of the language, and what is just syntactic sugar?
2. What are the rules for determining whether a given PVS expression is well typed?
3. How is subtyping handled, and in particular, how are proof obligations corresponding to subtypes generated?
4. What is the meaning, in set-theoretic terms, of a PVS expression or assertion?
5. Are the type rules sound with respect to the semantics?
6. Are the proof rules sound with respect to the semantics?
7. What is the form of dependent typing used by PVS, and what kinds of type dependencies are disallowed by the language?
8. What is the meaning of theory-level parametricity, and what, if any, are the semantic limits on such parameterization?
9. What language extensions are incompatible with the reference semantics given here?

Chapter 8 summarizes the answers to these questions.

1.1 Real versus Idealized PVS

The semantic treatment in this report is incomplete in some important ways. It does not treat the nonlogical parts of the language. In particular, it ignores arithmetic and recursive definitions. It also omits abstract datatypes [OS97]. These will be treated in a future expanded version.

The present semantics also makes several idealizations from the real PVS for the purpose of clarity. While the semantic treatment is not comprehensive, the idealization of PVS used here is faithful to the implemented form of PVS.

1. *No name resolution.* All names must be in fully resolved form with their theory name and actual parameters. We regard name resolution as a convenience provided by the PVS type checker and not an operation

with any semantic relevance. A technical description of name resolution in PVS will be given elsewhere.

2. *No overloading.* As with name resolution, overloading is a syntactic convenience with no semantic import.

3. *No* `IMPORTING`*s.* The importing of theories is a hint to name resolution. The semantic definition assumes that all instances of theories declared prior to the present one are visible.

4. *Variable declarations ignored.* All variables must be locally declared. Global variable declarations are regarded as a syntactic convenience.

5. *No records.* These are ignored in the semantic treatment since product types capture all the semantically essential features of records.

1.2 Semantic Preliminaries

The PVS specification language is based on higher-order logic. This means that variables can range over individuals (such as numbers) as well as functions, functions of functions, and so on. As is well known, some type distinction is needed; otherwise, it is easy to obtain a contradiction by defining the predicate $N(P)$ as $\neg(P(P))$ so that both $N(N)$ and $\neg N(N)$ hold. In the theory of types [Chu40], the universe is stratified into distinct types so that a predicate can be applied only to a *lower* type and thus cannot be applied to itself.

Types also serve as a powerful mechanism for detecting syntactic and semantic errors through typechecking. This role of types is best exemplified by their use in various programming languages such as Algol, Ada, and ML, and is also heavily emphasized in the PVS type system.

The desirability for strong typing in a specification logic is not widely accepted. Fraenkel *et al* [FBHL84] express the opinion that such typing is repugnant in a mathematical logic since it constrains expressiveness by not allowing individuals of differing types to be treated uniformly. Lamport [Lam94] argues that type correctness is like any other program property and should be established by means of a proof rather than by syntactic restraints. Lamport and Paulson [LP97] analyze the tradeoffs between typed and untyped specification languages. We claim that

1. Types impose a useful discipline on the specification.

2. Types lead to easy and early detection of a large class of syntactic and semantic errors.

3. Type information is useful in mechanized reasoning.

The semantics of a higher-order logic is given by mapping the well-formed types of the logic to *sets*, and the well-formed terms of the logic to *elements* of the sets representing their type. The set constructions we use can be formalized within Zermelo-Fraenkel set theory with the axiom of choice (ZFC). The intended interpretation of a function type in higher-order logic is that it represents the set of all functions from the set representing the domain type to the set representing the range types.[1] PVS also has predicate subtypes that are to be interpreted over the subsets of the set representing the parent type.

The semantics of PVS will be given by considering a sequence of increasingly expressive fragments of PVS. The semantics of each fragment of PVS will be presented in three steps. The first step is to define a set-theoretic universe containing enough sets to represent the PVS types. The second step is to define a typechecking operation that determines whether a given PVS expression is well typed. The third step is to define a semantic function that assigns a representation in the semantic universe to each well-typed PVS type and term.

We first lay out the ZFC set constructions needed for defining the semantics of PVS. The base types in PVS consist of the Booleans `bool` and the real numbers `real`. The Booleans can be modeled by any two-element set, say $\mathbf{2}$ consisting of the elements $\mathbf{0}$ and $\mathbf{1}$, where $\mathbf{0}$ is the empty set and the only element of the set $\mathbf{1}$. The real numbers can be captured by means of Dedekind cuts or Cauchy sequences, and we label this set $\mathbf{R}$.

To define the semantics, we need a universe that contains the sets $\mathbf{2}$ and $\mathbf{R}$ and is closed under Cartesian products (written as $X \times Y$) and power sets (written as $\wp(X)$). Note that functions are modeled as *graphs*, that is, sets of ordered pairs, so that a function type $[A \rightarrow B]$ is represented by a subset of the powerset $\wp(\llbracket A \rrbracket \times \llbracket B \rrbracket)$ of the Cartesian product of the sets $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$ representing A and B, respectively. A set F that is a subset of $X \times Y$ is the graph of a function with domain X and range Y if for every $x \in X$ there is a $y \in Y$ such that $\langle x, y \rangle \in F$, and whenever $\langle x, y \rangle \in F$ and $\langle x, y' \rangle \in F$, we have $y = y'$. For such a set F, $\mathit{Function}(F)$ holds and $\mathit{dom}(F) = X$. The set of graphs of total functions from a set Y to a set X is represented as X^Y.

[1] It is only in the *standard* model of higher-order logic that the function type is required to represent the set of all functions from the domain set to the range set. Higher-order logic can be interpreted in *general models* where the function type can be interpreted in any manner as long as it satisfies the various axioms such as application, abstraction, and extensionality [And86]. Higher-order logic is complete with respect to the general models interpretation so that a statement that is valid in all models is provable. It is, however, incomplete with respect to the standard model.

If F is the graph of a function and t an element in its domain, then $F(t)$ represents the result of applying the function F to t. At the semantic level, a function F will never be applied to an argument t outside $dom(F)$, because in the PVS language, a function application is typechecked so that the argument expression has the same type as the domain type of the function expression.

We can model the entire type universe of the simply typed fragment of PVS by the set U, which is defined cumulatively by starting from the base sets **2** and **R**, and including the Cartesian products, the function spaces, and subsets of previously included sets, at each stage. Cartesian products are used to model products in PVS, and function spaces model function types. Subsets are needed to model predicate subtypes. It is sufficient to iterate these stages up to the ordinal ω.

Definition 1.1 (type universe)

$$\begin{array}{rcl} U_0 & = & \{\mathbf{2}, \mathbf{R}\} \\ U_{i+1} & = & U_i \cup \{X \times Y \mid X, Y \in U_i\} \cup \{X^Y \mid X, Y \in U_i\} \cup \bigcup_{X \in U_i} \wp(X) \\ U_\omega & = & \bigcup_{i \in \omega} U_i \\ U & = & U_\omega \end{array}$$

■

We refer to U as the *basic universe.*[2] The semantic definitions below will assign a set in U to each PVS type and an element in $\bigcup U$ to each well-typed term of PVS. The *rank* of a set X in U is the least i such that $X \in U_i$. The notion of rank plays an important role in the semantics of dependent types and parametric theories.

1.3 Related Work

There is a long history of work in specification languages. Many ideas similar to those underlying the PVS specification language also occur in other specification languages.

The *wide-spectrum* languages are typically based on set theory or higher-order logic. The language VDM is one of the earliest such specification formalisms [Jon90]. It is based on a first-order logic with partial functions augmented with datatype axioms. The datatype theories in VDM include those

[2]The inclusion of X^Y in U is actually redundant but aids clarity.

for finite sets, maps, sequences, and recursive datatypes such as lists and trees. VDM has a notion of datatype invariants that yields a simple form of predicate subtyping. Operations on state are specified in terms of pre-condition/post-condition pairs. Specifications are structured into parameterized modules. In contrast to VDM, the PVS language is based on strictly typed higher-order logic with a built-in notion of predicate subtyping and dependent typing. The resulting PVS logic is more compact in that many of the datatypes that are presented axiomatically in VDM can be defined within PVS. There is no built-in notion of state in PVS since it is possible to use the higher-order logic of PVS to define a variety of state-based formalisms, including various linear and branching-time temporal logics. VDM uses a 3-valued logic for the logical connectives in order to deal with partial functions, whereas PVS uses a classical 2-valued logic and predicate subtyping to assign a type to a partial function as a total function on its domain of definition. Jones [Jon90] provides only an informal semantics for VDM. The RAISE system is a comprehensive toolset based on the ideas of VDM [RAISE92].

The Z specification language [Spi88] is another wide-spectrum language based on a typed first-order set theory. A Z specification is a collection of schemas consisting of declarations of types and constants accompanied with invariants. Z schemas can either specify datatype invariants or pre-condition/post-condition constraints. A schema calculus is used to combine schemas using logical connectives. Spivey [Spi88] presents a formal semantics for Z without giving a proof system or a soundness proof. Spivey's treatment of partial functions in the Z semantics employs the commonly used convention that $f(a)$ when a is not in the domain of a is some arbitrarily chosen value. This is fine for most purposes but can be confusing when dealing with recursively defined partial functions. For example, the definition $\mathit{bad}(x) = 1 + \mathit{bad}(x)$ is everywhere undefined but admitting it as an axiom leads to an immediate contradiction. Z also lacks any mechanism for conservative extensions such as definitional principles for constants and datatypes so that the consistency of Z specification has to be demonstrated by exhibiting a model.

Algebraic specification languages like OBJ [FGJM85] and Larch [GH93] provide an equational/rewriting framework for specifying datatypes and operations on datatypes. OBJ has many of the same theory parameterization mechanisms as PVS. The subsort mechanism in OBJ is also similar except that it is handled by introducing *retracts* or runtime checks rather than proof obligations generated by the type checker. The OBJ logic is quite restricted compared to PVS since it is based on a first-order, equational framework with an *initial* semantics where two ground terms are distinct unless they can be proved equal. OBJ has very limited support for proof development and is primarily intended as an executable specification language.

The specification languages that are closer to PVS are those that accompany various automated proof checking systems. The closest of these is EHDM [EHDM93], which employs a similar higher-order logic with subtyping and proof obligation generation. EHDM lacks many of the features of PVS: subtyping is restricted to type declarations and there is no dependent typing.

Higher-order logic is used by other systems such as HOL [GM93] and TPS [AMCP84]. Both HOL and TPS employ simply typed higher-order logic without features such as subtyping, dependent typing, or parametric theories. Andrews [And86] gives a thorough account of the semantic aspects of higher-order logic. The formal semantics of the HOL logic are carefully outlined (by Pitts) in the book by Gordon and Melham [GM93].

Systems like Coq [DFH$^+$91] and Nuprl [CAB$^+$86] are based on intuitionistic higher-order logics. Coq allows quantification over types, whereas Nuprl has quantification over a hierarchy of type universes. Both logics admit dependent typing. The set-theoretic semantics of dependently typed intuitionistic type theories has been studied by Dybjer [Dyb91] and Howe [How91, How96]. Not surprisingly, their semantic treatment of dependent typing is similar to the one given here but they do not delimit the possible dependencies as is done with the PVS semantics. The PVS semantics presented here clearly specifies the kind of type dependencies that are disallowed in the logic. Dybjer and Howe also do not address subtyping but do describe the semantics of language features missing in PVS (type universes in the case of Howe, and inductive families in the case of Dybjer). Dybjer does not identify the universe over which terms and types are interpreted. Howe requires an infinite sequence of inaccessible cardinals for his universe construction.

1.4 Outline

In Chapter 2, we define the syntax and semantics of the simply typed fragment of PVS. Type definitions are also introduced in this chapter along with the definition of definitional equivalence on types. Chapter 3 adds subtyping to the simply typed fragment and specifies the additional type rules and semantic definitions that are needed. Chapter 4 extends the language with dependent function and product types. Theories and parametric theories are introduced into the language in Chapter 5. The type rules and semantics for conditional expressions and the logical connectives defined using conditional expressions are introduced in Chapter 6. Chapter 7 specifies the axioms and inference rules of PVS.

Chapter 2

The Simple Type Theory

PVS is a strongly typed specification language. The simply typed fragment includes types constructed from the base types by the function and product type constructions, and expressions constructed from the constants and variables by means of application, abstraction, and tupling. Expressions are checked to be well typed under a *context*, which is a partial function that assigns a *kind* (one of `TYPE`, `CONSTANT`, or `VARIABLE`) to each symbol, and a type to the constant and variable symbols. We use the metavariables Γ, Δ, and Θ to range over contexts. The metavariables A, B, and T range over PVS type expressions, the metavariables r and s range over symbols (identifiers), the metavariables x and y range over PVS variables, and the metavariables a, b, f, and g range over PVS terms. Given a context Γ and a symbol s, we say that $\Gamma(s)$ is undefined if s is not declared in Γ.

The *pretypes* of the simple type theory include the *base* types such as `bool` and `real`. A *function* pretype from domain pretype A to range pretype B is constructed as $[A \rightarrow B]$. A *product* pretype of A_1, A_2 is constructed as $[A_1, A_2]$. A *type* is a pretype that has been typechecked in a given context. Types in the simple type theory are simple enough that the only distinction between pretypes and types is that the symbols in a type must be appropriately declared in the given context.

Example 2.1 (pretypes) `bool`, `real`, [`bool`, `real`], [[`real`, `bool`]$\rightarrow$`bool`]. ■

The *preterms* of the language consist of the constants, variables, pairs, projections, applications, and abstractions. The metavariables c and d range over constants. Pairs are of the form (a_1, a_2) where each a_i is a preterm. Applications have the form $f\ a$ where f and a are preterms. A pair projection is an expression of the form $\mathtt{p}_i\ a$, where $i \in \{1, 2\}$. Lambda abstractions have the form $\lambda(x{:}\,T){:}\,a$, where T is a pretype and a is a preterm. Parentheses are used for disambiguation. A *term* is a preterm that has been typechecked in a given context.

Example 2.2 (preterms) $\mathtt{TRUE}$, $\neg\ \mathtt{TRUE}$, $\lambda\ (x : \mathtt{bool}){:}\ \neg(x)$, $\mathsf{p}_2\ (\mathtt{TRUE}, \mathtt{FALSE})$, $(\mathtt{TRUE}, \lambda\ (x : \mathtt{bool}) : \neg\ (\neg\ x))$. ■

2.1 Contexts

A context is a sequence of declarations, where each declaration is either a type declaration s : TYPE, a constant declaration $c : T$ where T is a type, or a variable declaration x : VAR T. Preterms and pretypes are typechecked with respect to a given context. The empty context is represented as $\{\}$. The well-formedness rules for contexts are presented below. A context can also be applied as a partial function so that for a symbol s with declaration D, $(\Gamma, s{:}\,D)(s) = D$ and $(\Gamma, s{:}\,D)(r) = \Gamma(r)$ for $r \neq s$. If s is not declared in Γ, then $\Gamma(s)$ is undefined. If Γ is a context, then for any symbol s, the kind of the symbol s in Γ is given by $\mathit{kind}(\Gamma(s))$. If the kind of s in Γ is CONSTANT or VARIABLE, then the $\mathit{type}(\Gamma(s))$ is the type assigned to s in Γ.

Example 2.3 (context)
$\mathtt{bool} : \mathtt{TYPE},\ \ \mathtt{TRUE} : \mathtt{bool},\ \ \mathtt{FALSE} : \mathtt{bool},\ \ x : \mathtt{VAR}\ [[\mathtt{bool}, \mathtt{bool}] \rightarrow \mathtt{bool}]$ ■

2.2 Type Rules

The type rules for the simple type theory are given by a recursively defined partial function τ that assigns

1. A type $\tau(\Gamma)(a)$ to a preterm a that is well typed with respect to a context Γ.

2. The keyword TYPE as the result of $\tau(\Gamma)(A)$ when A is a well-formed type under context Γ.

3. The keyword CONTEXT as the result of $\tau(\Gamma)(\Delta)$ when Δ is a well-formed context under context Γ. The context Γ is empty for the simply typed fragment so that typechecking is always invoked as $\tau()(\Gamma)$.

Otherwise, τ is undefined in the case of an ill-typed preterm or an ill-formed type or context.

The type rules are given by the recursive definition for τ. Typechecking in PVS assigns a "canonical" type to a preterm. Customarily, type rules are presented as inference rules, but a functional presentation is more appropriate for PVS since

1. The type assignment is deterministic. A term can, in general, though not in the simply typed fragment, be assigned a number of types but it always has at most one canonical type.

2. The soundness proof need only show that the meaning of the term is an element of the meaning of its canonical type. Thus, only the canonical type derivation for a term has to be shown sound and not every valid type derivation.

3. The meaning of a term is therefore given by recursion on the term itself and not on its typing derivation. There is no need to show separately that this meaning is *coherent*, that is, independent of the typing derivation.

A functional presentation of the type rules also leads to natural and straightforward soundness arguments. Note that the well-formedness rules for contexts and types are trivial in the simply typed situation but become more meaningful when the type theory is extended. Note also that in the type rules for expressions and types, the well-formedness of the relevant context is not explicitly checked. These rules do preserve the well-formedness of the context in each recursive call so that if the initial context is well formed, then so is every intermediate one.

Definition 2.4 (type rules)

$$\begin{array}{rcl}
\tau()(\{\}) & = & \texttt{CONTEXT} \\
\tau()(\Gamma, s : \texttt{TYPE}) & = & \texttt{CONTEXT},\ \textit{if}\ \Gamma(s)\ \textit{is undefined} \\
 & & \textit{and}\ \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, c{:}\, T) & = & \texttt{CONTEXT},\ \textit{if}\ \Gamma(c)\ \textit{is undefined,} \\
 & & \tau(\Gamma)(T) = \texttt{TYPE}, \\
 & & \textit{and}\ \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau()(\Gamma, x{:}\, \texttt{VAR}\ T) & = & \texttt{CONTEXT},\ \textit{if}\ \Gamma(x)\ \textit{is undefined,} \\
 & & \tau(\Gamma)(T) = \texttt{TYPE}, \\
 & & \textit{and}\ \tau()(\Gamma) = \texttt{CONTEXT} \\
\tau(\Gamma)(s) & = & \texttt{TYPE},\ \textit{if}\ \textit{kind}(\Gamma(s)) = \texttt{TYPE} \\
\tau(\Gamma)([A{\to}B]) & = & \texttt{TYPE},\ \textit{if}\ \tau(\Gamma)(A) = \tau(\Gamma)(B) = \texttt{TYPE}
\end{array}$$

$$
\begin{array}{rcl}
\tau(\Gamma)([A_1, A_2]) & = & \texttt{TYPE},\ \textit{if}\ \tau(\Gamma)(A_i) = \texttt{TYPE}\ \textit{for}\ 1 \leq i \leq 2 \\
\tau(\Gamma)(s) & = & \textit{type}(\Gamma(s)), \\
 & & \ \textit{if kind}(\Gamma(s)) \in \{\texttt{CONSTANT}, \texttt{VARIABLE}\} \\
\tau(\Gamma)(f\ \ a) & = & B,\ \textit{if}\ \tau(\Gamma)(f) = [A \rightarrow B]\ \textit{and}\ \tau(\Gamma)(a) = A \\
\tau(\Gamma)(\lambda(x{:}\,T){:}\,a) & = & [T \rightarrow \tau(\Gamma, x{:}\,\texttt{VAR}\ T)(a)],\ \textit{if}\ \Gamma(x)\ \textit{is undefined} \\
 & & \textit{and}\ \tau(\Gamma)(T) = \texttt{TYPE} \\
\tau(\Gamma)((a_1, a_2)) & = & [\tau(\Gamma)(a_1), \tau(\Gamma)(a_2))] \\
\tau(\Gamma)(\texttt{p}_i\ \ a) & = & T_i,\ \textit{where} \\
 & & \tau(\Gamma)(a) = [T_1, T_2]
\end{array}
$$

■

In the type rule for lambda abstraction, the constraint that $\Gamma(x)$ must be undefined can be satisfied by suitably renaming the bound variable since we treat terms as equivalent modulo the renaming of bound variables.

Example 2.5 (type rules) Let Ω label the context `bool` : `TYPE`, `TRUE` : `bool`, `FALSE` : `bool`

$$
\begin{array}{rcl}
\tau()(\{\}) & = & \texttt{CONTEXT} \\
\tau()(\Omega) & = & \texttt{CONTEXT} \\
\tau(\Omega)([[\texttt{bool}, \texttt{bool}] \rightarrow \texttt{bool}]) & = & \texttt{TYPE} \\
\tau(\Omega)((\texttt{TRUE}, \texttt{FALSE})) & = & [\texttt{bool}, \texttt{bool}] \\
\tau(\Omega)(\texttt{p}_2(\texttt{TRUE}, \texttt{FALSE})) & = & \texttt{bool} \\
\tau(\Omega)(\lambda(x : \texttt{bool}){:}\ \texttt{TRUE}) & = & [\texttt{bool} \rightarrow \texttt{bool}]
\end{array}
$$

■

2.3 Semantics

Recall that a preterm a with a type assigned by τ under context Γ is said to be a term of type $\tau(\Gamma)(a)$ in the context Γ. If γ is an *assignment* for the symbols declared in context Γ, the semantics of the simple type theory of PVS is given by mapping a type T to a (possibly empty) set $\mathcal{M}(\Gamma \mid \gamma)(T)$, and a term a with assigned type T to an element of the set $\mathcal{M}(\Gamma \mid \gamma)(T)$ in the basic universe U. The assignment γ is a list of bindings of the form $\{s_1 \leftarrow t_1\} \ldots \{s_n \leftarrow t_n\}$. The application of an assignment γ to a symbol s is such that $\gamma\{s \leftarrow t\}(s)$ is t, whereas $\gamma\{r \leftarrow t\}(s)$ is $\gamma(s)$ when $r \not\equiv s$.

The meaning function $\mathcal{M}$ returns the meaning of a well-formed type A and a well-formed expression a in the context Γ under an assignment γ as $\mathcal{M}(\Gamma \mid \gamma)(A)$ and $\mathcal{M}(\Gamma \mid \gamma)(a)$,respectively. The meanings of type names, constants, and variables declared in Γ are obtained from the assignment γ. A function type is mapped to the corresponding function space. A product type is mapped to the corresponding Cartesian product. An application term is interpreted by means of set-theoretic function application. A lambda abstraction yields the graph of the corresponding function. A pair expression is mapped to the corresponding set-theoretic ordered pair.

Definition 2.6 (meaning function)

$$\begin{array}{rcl}
\mathcal{M}(\Gamma \mid \gamma)(s) & = & \gamma(s), \\
 & & \quad \textit{if } \textit{kind}(\Gamma(s)) \in \{\texttt{TYPE}, \texttt{CONSTANT}, \texttt{VARIABLE}\} \\
\mathcal{M}(\Gamma \mid \gamma)([A{\rightarrow}B]) & = & \mathcal{M}(\Gamma \mid \gamma)(B)^{\mathcal{M}(\Gamma \mid \gamma)(A)} \\
\mathcal{M}(\Gamma \mid \gamma)([T_1, T_2]) & = & \mathcal{M}(\Gamma \mid \gamma)(T_1) \times \mathcal{M}(\Gamma \mid \gamma)(T_2) \\
\mathcal{M}(\Gamma \mid \gamma)(f\ a) & = & (\mathcal{M}(\Gamma \mid \gamma)(f))(\mathcal{M}(\Gamma \mid \gamma)(a)) \\
\mathcal{M}(\Gamma \mid \gamma)(\lambda(x{:}\,T){:}\,a) & = & \{\langle y, z\rangle \mid \ y \in \mathcal{M}(\Gamma \mid \gamma)(T), \\
 & & \qquad\qquad z = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ T \mid \gamma\{x \leftarrow y\})(a)\} \\
\mathcal{M}(\Gamma \mid \gamma)((a_1, a_2)) & = & \langle \mathcal{M}(\Gamma \mid \gamma)(a_1), \mathcal{M}(\Gamma \mid \gamma)(a_2)\rangle \\
\mathcal{M}(\Gamma \mid \gamma)(\mathtt{p}_i\ a) & = & t_i, \ \textit{where } \mathcal{M}(\Gamma \mid \gamma)(a) = \langle t_1, t_2\rangle
\end{array}$$

■

Example 2.7 (meaning function) Let ω be an assignment for the context Ω in Example 2.5, of the form

$$\{\texttt{bool} \leftarrow \mathbf{2}\}\{\texttt{TRUE} \leftarrow \mathbf{1}\}\{\texttt{FALSE} \leftarrow \mathbf{0}\}$$

then

$$\begin{array}{rcl}
\mathcal{M}(\Omega \mid \omega)([\texttt{bool}, \texttt{bool}]) & = & \mathbf{2} \times \mathbf{2} \\
\mathcal{M}(\Omega \mid \omega)((\texttt{TRUE}, \texttt{FALSE})) & = & \langle \mathbf{1}, \mathbf{0}\rangle \\
\mathcal{M}(\Omega \mid \omega)(\lambda(x{:}\,\texttt{bool}){:}\,\texttt{TRUE}) & = & \{\langle \mathbf{0}, \mathbf{1}\rangle, \langle \mathbf{1}, \mathbf{1}\rangle\}
\end{array}$$

■

Definition 2.8 (satisfaction) *A context assignment γ is said to* satisfy *a context Γ (in symbols $\gamma \models \Gamma$) iff*

1. $\gamma(\texttt{bool}) = \mathbf{2}$,

2. $\gamma(\texttt{TRUE}) = \mathbf{1}$,

3. $\gamma(\texttt{FALSE}) = \mathbf{0}$,

4. $\gamma(s) \in U$ *whenever* $\mathit{kind}(\Gamma(s)) = \texttt{TYPE}$, *and*

5. $\gamma(s) \in \mathcal{M}(\Gamma \mid \gamma)(\mathit{type}(\Gamma(s)))$
   whenever $\mathit{kind}(\Gamma(s)) \in \{\texttt{CONSTANT}, \texttt{VARIABLE}\}$.

∎

Example 2.9 (satisfaction)

1. The assignment ω satisfies context Ω.

2. The assignment $\omega\{\texttt{one} \leftarrow \mathbf{1}\}\{\texttt{zero} \leftarrow \mathbf{0}\}$ satisfies the context Ω, $\texttt{one} : \texttt{TYPE}$, $\texttt{zero} : \texttt{one}$.

∎

We need one useful proposition that asserts that typing judgements are not invalidated when the context is extended.

Proposition 2.10 *If* $\tau()(\Gamma) = \tau()(\Gamma') = \texttt{CONTEXT}$ *and* Γ *is a prefix of* Γ', *then for all pretypes* A, $\tau(\Gamma)(A) = \texttt{TYPE}$ *implies* $\tau(\Gamma')(A) = \texttt{TYPE}$, *and for all preterms* a, $\tau(\Gamma)(a) = A$ *implies* $\tau(\Gamma')(a) = A$.

The following theorems follow from the induction suggested by the definitions of τ and $\mathcal{M}$. The first of these is straightforward and is given without proof.

Theorem 2.11 (type construction) *If* $\tau()(\Gamma) = \texttt{CONTEXT}$ *and* $\tau(\Gamma)(a) = A$, *then* $\tau(\Gamma)(A) = \texttt{TYPE}$.

Theorem 2.12 (type soundness) *If* $\tau()(\Gamma) = \texttt{CONTEXT}$, γ *satisfies* Γ, *and* $\tau(\Gamma)(A) = \texttt{TYPE}$, *then* $\mathcal{M}(\Gamma \mid \gamma)(A) \in U$.

Proof. The proof is by induction on the structure of the pretype A. Recall that if $X \in U$, then for some i, $X \in U_i$. This yields three cases:

1. $A \equiv s$: By Definition 2.4, $\Gamma(s)$ is defined and $\mathit{kind}(\Gamma(s))$ is $\texttt{TYPE}$. Then by Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)(s)$ is $\gamma(s)$, and by Definition 2.8, $\gamma(s) \in U$.

2. $A \equiv [B \to C]$: We then have that $\tau(\Gamma)(B) = \tau(\Gamma)(C) = \texttt{TYPE}$. Letting X label $\mathcal{M}(\Gamma \mid \gamma)(B)$, and Y label $\mathcal{M}(\Gamma \mid \gamma)(C)$, we have by the induction hypothesis that $X \in U$ and $Y \in U$. Let j be the least rank such that $\mathcal{M}(\Gamma \mid \gamma)(B) \in U_j$ and $\mathcal{M}(\Gamma \mid \gamma)(C) \in U_j$. By Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)(A) = Y^X$, and hence $\mathcal{M}(\Gamma \mid \gamma)(A) \in U_{j+1}$ by Definition 1.1.

3. $A \equiv [A_1, A_2]$: Again by Definition 2.4 and the induction hypothesis, we have for each $i \in \{1, 2\}$, that $\mathcal{M}(\Gamma \mid \gamma)(A_i) \in U$. Let j be the least rank such that for $i \in \{1, 2\}$, $\mathcal{M}(\Gamma \mid \gamma)(A_i) \in U_j$. Then, it is easy to verify from Definition 1.1 that $\mathcal{M}(\Gamma \mid \gamma)(A) \in U_{j+1}$.

■

Theorem 2.13 (term soundness) *If* $\tau()(\Gamma) = \texttt{CONTEXT}$*,* γ *satisfies* Γ*, and* $\tau(\Gamma)(a)$ *is defined and equal to* A*, then* $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$*.*

Proof. The proof is by induction on the structure of preterms.

1. $a \equiv s$: By Definition 2.4, we have that $\mathit{type}(\Gamma(s)) = A$. By Definitions 2.6 and 2.8, we have that $\mathcal{M}(\Gamma \mid \gamma)(a) = \gamma(s)$ and $\gamma(s) \in \mathcal{M}(\Gamma \mid \gamma)(A)$.

2. $a \equiv (f\ b)$: By Definition 2.4, $\tau(\Gamma)(f) = [B \to A]$, and $\tau(\Gamma)(b) = B$, for some B such that $\tau(\Gamma)(B) = \texttt{TYPE}$. Let $\mathcal{M}(\Gamma \mid \gamma)(A)$ be X and $\mathcal{M}(\Gamma \mid \gamma)(B)$ be Y, then by Definitions 2.4 and 2.6, and the induction hypothesis, we have $\mathcal{M}(\Gamma \mid \gamma)(f) \in X^Y$ and $\mathcal{M}(\Gamma \mid \gamma)(b) \in Y$. It therefore follows by Definition 2.6 that $\mathcal{M}(\Gamma \mid \gamma)((f\ b)) = (\mathcal{M}(\Gamma \mid \gamma)(f))(\mathcal{M}(\Gamma \mid \gamma)(b))$, and hence $\mathcal{M}(\Gamma \mid \gamma)((f\ b)) \in X$.

3. $a \equiv (\lambda(x{:}\,C){:}\,b)$: By Definition 2.4, we have that $\tau(\Gamma)(a)$ is $[C \to B]$, where $\tau(\Gamma, x{:}\,\texttt{VAR}\ C)(b)$ is B. Let X be $\mathcal{M}(\Gamma \mid \gamma)(C)$, and Y be $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ C \mid \gamma\{x \leftarrow u\}))(B)$. By the induction hypothesis, we have that for any $u \in Y$, $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ C \mid \gamma\{x \leftarrow u\})(b) \in X$. Since $\mathcal{M}(\Gamma \mid \gamma)(a)$ is $\{\langle u, v\rangle \mid u \in X, v = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ C \mid \gamma\{x \leftarrow u\})(b)\}$, we have that $\mathcal{M}(\Gamma \mid \gamma)(a) \in X^Y$.

4. $a \equiv (a_1, a_2)$: By Definition 2.4, $\tau(\Gamma)(a) = [A_1, A_2]$, where $\tau(\Gamma)(a_i) = A_i$ for $i \in \{1, 2\}$. By the induction hypothesis, $\mathcal{M}(\Gamma \mid \gamma)(a_i) \in \mathcal{M}(\Gamma \mid \gamma)(A_i)$ for $i \in \{1, 2\}$. By Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)(a) = \langle \mathcal{M}(\Gamma \mid \gamma)(a_1), \mathcal{M}(\Gamma \mid \gamma)(a_2)\rangle$ and hence $\mathcal{M}(\Gamma \mid \gamma)(a)$ is an element of $\mathcal{M}(\Gamma \mid \gamma)(A)$ which is $\mathcal{M}(\Gamma \mid \gamma)(A) \times \mathcal{M}(\Gamma \mid \gamma)(A_n)$.

5. $a \equiv \mathtt{p}_i\ b$: In this case, we know by Definition 2.4 that $\tau(\Gamma)(b) = [A_1, A_2]$ with $i \in \{1, 2\}$, and $\tau(\Gamma)(a) = A_i$. By the induction hypothesis, $\mathcal{M}(\Gamma \mid \gamma)(b) = \langle t_1, t_2 \rangle$, and by Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)(a) = t_i$ and $\mathcal{M}(\Gamma \mid \gamma)(\tau(\Gamma)(b)) = \mathcal{M}(\Gamma \mid \gamma)(A_1) \times \mathcal{M}(\Gamma \mid \gamma)(A_2)$, hence $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A_i)$.

∎

These three theorems (2.11, 2.12, and 2.13) are the key invariants that must be satisfied by the semantics when the language is extended below with type definitions, subtypes, dependent types, and parametric theories.

2.4 Some Syntactic Operations

We first define the operation of collecting the free variables of a term a in a given context Γ as $FV(\Gamma)(a)$, and then define the operation of substitution.

Definition 2.14 (free variables)

$$
\begin{array}{rcl}
FV(\Gamma)(s) & = & \begin{cases} \{s\}, & \textit{if kind}(\Gamma(s)) = \mathtt{VARIABLE} \\ \emptyset, & \textit{otherwise} \end{cases} \\
FV(\Gamma)(f\ a) & = & FV(\Gamma)(f) \cup FV(\Gamma)(a) \\
FV(\Gamma)(\lambda(x\colon T)\colon a) & = & FV(\Gamma, x\colon \mathtt{VAR}\ T)(a) - \{x\} \\
FV(\Gamma)((a_1, a_2)) & = & FV(\Gamma)(a_1) \cup FV(\Gamma)(a_2) \\
FV(\Gamma)(\mathtt{p}_i\ a) & = & FV(\Gamma)(a)
\end{array}
$$

∎

Definition 2.15 (substitution)

$$
\begin{array}{rcl}
s[a_1/x_1, \ldots, a_n/x_n] & = & \begin{cases} a_i, & \textit{if for some minimal } i, s \equiv x_i \\ s, & \textit{otherwise} \end{cases} \\
(f\ a)[a_1/x_1, \ldots, a_n/x_n] & = & (f[a_1/x_1, \ldots, a_n/x_n] \\
 & & \qquad a[a_1/x_1, \ldots, a_n/x_n]) \\
(\lambda(y\colon T)\colon a)[a_1/x_1, \ldots, a_n/x_n] & = & (\lambda(y'\colon T)\colon\ a[y'/y, a_1/x_1, \ldots, a_n/x_n]), \\
 & & \textit{where } y' \textit{ is a fresh variable} \\
(b_1, b_2)[a_1/x_1, \ldots, a_n/x_n] & = & (b_1[a_1/x_1, \ldots, a_n/x_n], \\
 & & \qquad b_2[a_1/x_1, \ldots, a_n/x_n]) \\
(\mathtt{p}_i\ a)[a_1/x_1, \ldots, a_n/x_n] & = & (\mathtt{p}_i\ a[a_1/x_1, \ldots, a_n/x_n])
\end{array}
$$

∎

Recall that terms are treated as syntactically equivalent modulo alpha conversion. The above definitions must be extended as more features are added to the language.

2.5 Type Definitions

Here we enrich contexts so that type symbols may have definitions. PVS does not allow recursive type definitions[1] so a type declaration/definition in a context may use only the symbols declared in the prior part of the context. The main difference in the extended language is that type names can have definitions. In such cases, the definitions rather than the type names are used to determine the actual type of an expression. In other words, two type expressions are treated as the same if they are *definitionally equivalent*. Most other specification languages tend to employ the weaker notion of *name equivalence* where syntactically different types are treated as distinct even when their definitions coincide.

To accommodate type definitions, a context can contain type declarations of the form s : `TYPE` $= T$, where T is a type. If context Γ contains such a declaration for s, then $\mathit{definition}(\Gamma(s))$ is T. To extend τ to handle type definitions under definitional equivalence, we must ensure that τ returns the canonical form of a type where all defined types have been replaced by their definitions. The operation $\delta(\Gamma)(T)$ returns the expanded form of a type relative to the context Γ.

Definition 2.16 (expanded type)

$$\begin{array}{rcl}
\delta(\Gamma)(s) & = & s,\ \textit{if definition}(\Gamma(s))\ \textit{is empty} \\
\delta(\Gamma)(s) & = & \delta(\Gamma)(\mathit{definition}(\Gamma(s))),\ \textit{if definition}(\Gamma(s))\ \textit{is nonempty} \\
\delta(\Gamma)([A{\rightarrow}B]) & = & [\delta(\Gamma)(A){\rightarrow}\delta(\Gamma)(B)] \\
\delta(\Gamma)([T_1, T_2]) & = & [\delta(\Gamma)(T_1), \delta(\Gamma)(T_2)]
\end{array}$$

■

The typing rules are augmented to return the type in expanded form. The main issue here is to determine that the definition part of a type declaration in a context is well formed relative to the preceding context. We also need to ensure that τ returns the expanded form of the type corresponding to a preterm.

[1]For the moment, we are not considering the PVS `DATATYPE` mechanism, which is a form of recursive type definition [OS97]. Recursive datatypes in the context of the HOL proof checking system are described by Melham [Mel89].

Definition 2.17 (type rules with type definitions)

$$\begin{array}{rcl}
\tau()(\Gamma, s : \mathtt{TYPE} = T) & = & \mathtt{CONTEXT},\ \textit{if}\ \Gamma(s)\ \textit{is undefined},\\
& & \tau()(\Gamma) = \mathtt{CONTEXT},\\
& & \textit{and}\ \tau(\Gamma)(T) = \mathtt{TYPE}\\
\tau(\Gamma)(s) & = & \delta(\Gamma)(\textit{type}(\Gamma(s))),\\
& & \ \textit{if}\ \textit{kind}(\Gamma(s)) \in \{\mathtt{CONSTANT}, \mathtt{VARIABLE}\}
\end{array}$$

■

Note that the δ operator is idempotent, and $\tau(\Gamma)(a)$ for a term a always returns an expanded type, that is, $\delta(\tau(\Gamma)(a)) = \tau(\Gamma)(a)$.

We do not need to update the definition of $\mathcal{M}$ from Definition 2.6 since the syntax for terms is unchanged, but we do need to revise the notion of a satisfying context assignment (from Definition 2.8) to respect the type definitions.

Definition 2.18 (satisfaction with type definitions) *An assignment γ satisfies a context Γ if in addition to the conditions in Definition 2.8, whenever $\textit{kind}(\Gamma(s)) = \mathtt{TYPE}$ and $\textit{definition}(\Gamma(s))$ (abbreviated as T) is nonempty, then $\gamma(s) = \mathcal{M}(\Gamma \mid \gamma)(T)$.* ■

Theorems 2.11 and 2.12 and 2.13 continue to hold under these extensions, and the proofs are easily adapted to the modified definitions.

Example 2.19 (type definition) Let Ω' be the context $\Omega, \mathtt{boolop}\colon \mathtt{TYPE} = [[\mathtt{bool}, \mathtt{bool}] \rightarrow \mathtt{bool}], \vee\colon \mathtt{boolop}$. Then

$$\begin{array}{rcl}
\tau()(\Omega') & = & \mathtt{CONTEXT}\\
\delta(\Omega')(\mathtt{boolop}) & = & [[\mathtt{bool}, \mathtt{bool}] \rightarrow \mathtt{bool}],\\
\tau(\Omega')(\vee) & = & [[\mathtt{bool}, \mathtt{bool}] \rightarrow \mathtt{bool}]
\end{array}$$

■

2.6 Summary

We have defined the simply typed fragment of PVS by introducing the syntax for pretypes and preterms, the type rules and semantics for well-formed contexts, types, and terms. The type rules are presented in a novel functional style where each well-formed context is assigned the label `CONTEXT`, each well-formed type is assigned the label `TYPE`, and each well-formed term is assigned a canonical type. The semantics takes a satisfying assignment for a context and maps a well-formed type to a set and a well-formed term to an element of the set corresponding to its canonical type. We then defined the syntactic operations of collecting the free variables in an expression and for substituting terms for variables in an expression.

The simple type theory is then extended with type definitions. With this extension, two type expressions are treated as equivalent if they are identical after all type definitions have been expanded. The operation δ returns the expanded form of a given type expression.

Chapter 3

Adding Subtypes

Subtyping is one of the main features of the PVS specification language.[1] Subtyping in PVS corresponds to the set-theoretic notion of a subset. It raises several delicate issues that were absent in the language presented thus far. In the simply typed fragment, each type corresponds to a set of values that is somehow structurally different from the set of values for another type so that a term has at most one type. Subtyping makes it possible to introduce the natural numbers as a subtype of the reals, and to treat the primes, the even numbers, and the odd numbers as subtypes of the natural numbers. With subtyping, a term can obviously have several possible types, but the type-checking function τ may return only a single type. We constrain τ to return a natural canonical type of an expression that is given by the declarations of the symbols in the expression. If the expression is used in a context where the expected type is a supertype of its canonical type, then the type correctness is straightforward. If the expected type is a subtype that is *compatible* with the canonical type of the expression, then typechecking generates proof obligations asserting that the expression satisfies the predicate constraints imposed by the expected type. Two types are compatible if they have equivalent maximal supertypes. Type equivalence in the presence of subtypes is not a simple notion. Subtyping also introduces the possibility of types being empty. Typed lambda calculi with possibly empty types have been studied by Meyer, Mitchell, Moggi, and Statman [MMMS90]. This chapter introduces predicate subtypes and defines the notions of compatibility and type equivalence prior to presenting the type rules and semantics.

We restrict our attention to contexts Γ that extend the declarations:

```
bool : TYPE,
```

[1]The form of subtyping used in PVS is derived from a suggestion of Friedrich von Henke.

```
TRUE : bool,
FALSE : bool,
boolop : [[bool, bool]→bool],
¬ : [bool→bool],
∨ : boolop,
∧ : boolop,
⊃ : boolop
```

We will abuse PVS notation to employ the customary infix forms of operations like $\vee$, $\wedge$, and $\supset$. The pretype corresponding to a predicate subtype has the form $\{x\colon T \mid a\}$ where x is a symbol, T is a pretype, and a is a preterm. A *predicate type* in PVS is a function type where the range is the primitive type `bool`. A predicate is a term that has a predicate type. If a is a term of type `bool`, then we can define the subtype $\{x\colon T \mid a\}$ consisting of those elements e of T satisfying $a[e/x]$ (e substituted for x in a). Since the elements of the subtype $\{x\colon T \mid a\}$ satisfy the predicate $\lambda(x\colon T)\colon a$, we call this type a *predicate subtype* to distinguish it from other forms of subtyping. Universal quantification $\forall(x\colon T)\colon a$ is just an abbreviation for the term $(\lambda(x\colon T)\colon a) = (\lambda(x\colon T)\colon \texttt{TRUE})$. Although we use the equality predicate in the definition of universal quantification and in the definitions below, the actual introduction of equality is deferred to a later section following the introduction of parametric theories. The equality between PVS terms of function type is to be interpreted as extensional equality. Note that the '=' symbol is used both for the formal equality symbol in the language and for metatheoretic equality.

Our first step will be to define the notion of a *maximal supertype* of a given type as $\mu(T)$. A *maximal type* T is one such that $\mu(T) = T$. In a given context, we will apply μ only to the expanded form (given by δ) of a type expression.

Definition 3.1 (maximal supertype)

$$\begin{aligned}
\mu(s) &= s\\
\mu(\{x\colon T \mid a\}) &= \mu(T)\\
\mu([A\rightarrow B]) &= [A\rightarrow \mu(B)]\\
\mu([A_1, A_2]) &= [\mu(A_1), \mu(A_2)]
\end{aligned}$$

■

Note that since subtypes correspond to subsets, in taking the maximal supertype of a function type, the domain type is held fixed. In most type theories with subtypes, the rule for subtyping between function types $[A\rightarrow B]$ and $[A'\rightarrow B']$ requires showing that A' is a subtype of A, and B is a subtype of B'.

Subtyping between function types is therefore said to be contravariant in the domain type and covariant in the range type. Subtyping on function types in PVS is covariant in the range type but is neither covariant nor contravariant in the domain type. This means that the function type $[\texttt{nat}\to\texttt{nat}]$ is not a supertype of the function type $[\texttt{int}\to\texttt{nat}]$. Such a subtyping relation would violate *extensionality*. Two functions on `nat` are extensionally equal when they return equal values when applied to equal arguments in `nat`. Consider two functions in $[\texttt{nat}\to\texttt{nat}]$: *abs* which returns the absolute value, and *idnat* which behaves as an identity function on natural numbers and returns 0 otherwise. These two functions will be erroneously identified if they can be viewed as being of type $[\texttt{nat}\to\texttt{nat}]$, and the subset interpretation of subtypes would be lost.

We will also employ a weaker supertype $\mu_0(T)$ or the *direct supertype*, that only considers supertypes of explicitly given subtypes of the form $\{x{:}\,T \mid a\}$.

Definition 3.2 (direct supertype)

$$\begin{array}{rcl}\mu_0(\{x{:}\,T \mid a\}) & = & \mu_0(T)\\ \mu_0(T) & = & T,\ \textit{otherwise}\end{array}$$

■

Example 3.3 (maximal supertype) Given a context containing the declarations

$$\begin{array}{l}\texttt{int}{:}\,\texttt{TYPE},\\ \texttt{0}{:}\,\texttt{int},\\ \le{:}\,[[\texttt{int},\texttt{int}]\to\texttt{bool}],\\ \texttt{nat}{:}\,\texttt{TYPE} = \{i{:}\,\texttt{int} \mid 0 \le i\}\\ \texttt{natinjection}{:}\,\texttt{TYPE} = \{f{:}\,[\texttt{nat}\to\texttt{nat}] \mid \forall(i,j{:}\,\texttt{nat}){:}\,f(i)=f(j)\supset i=j\}\end{array}$$

we have

$$\begin{array}{rcl}\mu(\texttt{natinjection}) & = & \mu([\texttt{nat}\to\texttt{nat}])\\ & = & [\texttt{nat}\to\mu(\texttt{nat})]\\ & = & [\texttt{nat}\to\texttt{int}]\\ \mu_0(\texttt{natinjection}) & = & [\texttt{nat}\to\texttt{nat}]\end{array}$$

■

Note that $\mu(\mu(A)) = \mu(A)$. Note also that a maximal supertype is never a subtype. We can in fact collect the predicates that constrain a type A relative to its maximal supertype $\mu(A)$ as $\pi(A)$.

Definition 3.4 (subtype constraints)

$$\begin{array}{rcl}
\pi(s) & = & \lambda(x\colon s)\colon \mathtt{TRUE} \\
\pi(\{y\colon T \mid a\}) & = & \lambda(x\colon \mu(T))\colon (\pi(T)(x) \wedge a[x/y]) \\
\pi([A \to B]) & = & \lambda(x\colon [A \to \mu(B)])\colon (\forall (y\colon A)\colon \pi(B)(x(y))) \\
\pi([A_1, A_2]) & = & \lambda(x\colon [\mu(A_1), \mu(A_2)])\colon (\pi(A_1)(\mathtt{p}_1\ x) \wedge \pi(A_2)(\mathtt{p}_2\ x))
\end{array}$$

■

Observe that in Definition 3.4, if $\tau(\Gamma)(A) = \mathtt{TYPE}$, then $\tau(\Gamma)(\pi(A)) = [\mu(A) \to \mathtt{bool}]$.[2]

Example 3.5 (subtype constraints)

$$\begin{array}{l}
\pi(\mathtt{nat}) \\
\quad = \ \lambda(j\colon \mathtt{int})\colon 0 \leq j \\
\pi([\mathtt{nat} \to \mathtt{nat}]) \\
\quad = \ \lambda(g\colon [\mathtt{nat} \to \mathtt{int}])\colon \forall (i\colon \mathtt{nat})\colon (\lambda(j\colon \mathtt{int})\colon 0 \leq j)(g(i)) \\
\pi(\mathtt{natinjection}) \\
\quad = \ \lambda(f\colon [\mathtt{nat} \to \mathtt{int}])\colon \qquad \pi([\mathtt{nat} \to \mathtt{nat}])(f) \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \ \ (\forall (i, j\colon \mathtt{nat})\colon f(i) = f(j) \supset i = j) \\
\quad = \ \lambda(f\colon [\mathtt{nat} \to \mathtt{int}])\colon \\
\qquad\qquad\qquad\quad (\lambda(g\colon [\mathtt{nat} \to \mathtt{int}])\colon \forall (i\colon \mathtt{nat})\colon (\lambda(j\colon \mathtt{int})\colon 0 \leq j)(g(i)))(f) \\
\qquad\qquad\quad \wedge \ \ (\forall (i, j\colon \mathtt{nat})\colon f(i) = f(j) \supset i = j)
\end{array}$$

■

Observe that $\pi(\mu(A))$ is essentially equivalent to $\lambda(x\colon \mu(A))\colon \mathtt{TRUE}$.

Since the subtype $\{x\colon T \mid p(x) \wedge q(x)\}$ can also be written as $\{x\colon T \mid q(x) \wedge p(x)\}$, we need a notion of equivalence between types. One way to do this is to make types "first-class" and to allow explicit theorems to be proved about type equivalence and subtyping. Since this would be a fairly drastic extension to the specification language, we have designed the PVS type system so as to avoid

[2]This is somewhat tricky in the case of $\pi(\{y\colon T \mid a\})$ since in $a[x/y]$, x has type $\mu(T)$, whereas y has type T. As shown in Chapter 6, the type rules for conjunction are such that $\tau(\Gamma, x\colon \mathtt{VAR}\ \mu(T))(\pi(T)(x) \wedge a)$ reduces to $\tau(\Gamma, x\colon \mathtt{VAR}\ \mu(T))(\pi(T)(x))$ and $\tau(\Gamma, x\colon \mathtt{VAR}\ \mu(T), \pi(T)(x))(a[x/y])$ where the first conjunct is added to the context as a contextual assumption. One can then show by induction that $\tau(\Gamma, x\colon \mathtt{VAR}\ \mu(T), \pi(T)(x))(a[x/y]) = \tau(\Gamma, y\colon \mathtt{VAR}\ T)(a)$.

any first-class treatment of types. It turns out that all the needed properties about types (such as equality and subtyping) can be obtained by generating ordinary proof obligations rather than by explicitly proving theorems about types.

We introduce below a metatheoretic operation that generates the proof obligations needed to establish that two (maximal) types are equivalent. This equivalence is denoted by $\simeq$ and is applied only to maximal types and returns a list of the proof obligations that must be proved. Note the invariant in the definition below that the arguments to $\simeq$ are always maximal. The definition of $\simeq$ makes use of the PVS equality predicate that will be introduced later. A list of formulas is represented as $a_1, \ldots, a_n$. Given two such lists $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, the concatenation of these two lists is written as $a_1, \ldots, a_m \; ; \; b_1, \ldots, b_n$.

Definition 3.6 (type equivalence proof obligations)

$$\begin{array}{rcl} (s \simeq s) & = & \texttt{TRUE} \\ ([A \rightarrow B] \simeq [A' \rightarrow B']) & = & ((\mu(A) \simeq \mu(A')); (\pi(A) = \pi(A')); (B \simeq B'))^{3} \\ ([A_1, A_2] \simeq [B_1, B_2]) & = & ((A_1 \simeq B_1); (A_2 \simeq B_2)) \\ (A \simeq B) & = & \texttt{FALSE}, \; \textit{otherwise} \end{array}$$

■

Example 3.7 (type equivalence) Building on the context given in Example 3.3, if we have the following variants of `nat` and `natinjection`:

$\texttt{NAT}\colon \texttt{TYPE} = \{i\colon \texttt{int} \mid i \leq \texttt{0} \;\supset\; i = 0\}$
$\texttt{NATinjection}\colon \texttt{TYPE} = \{f\colon [\texttt{NAT}\rightarrow\texttt{NAT}] \mid \forall (i, j\colon \texttt{NAT})\colon f(i) = f(j) \;\supset\; i = j\}$

we get

$$\mu([\texttt{natinjection}\rightarrow\texttt{natinjection}]) = [\texttt{natinjection}\rightarrow[\texttt{nat}\rightarrow\texttt{int}]]$$
$$\mu([\texttt{NATinjection}\rightarrow\texttt{NATinjection}]) = [\texttt{NATinjection}\rightarrow[\texttt{NAT}\rightarrow\texttt{int}]]$$

$$\begin{array}{l} \mu([\texttt{natinjection}\rightarrow\texttt{natinjection}]) \simeq \mu([\texttt{NATinjection}\rightarrow\texttt{NATinjection}]) \\ \quad = \quad (\mu(\texttt{natinjection}) \simeq \mu(\texttt{NATinjection})); \\ \qquad\quad (\pi(\texttt{natinjection}) = \pi(\texttt{NATinjection})); \\ \qquad\quad ([\texttt{nat}\rightarrow\texttt{int}] \simeq [\texttt{NAT}\rightarrow\texttt{int}]) \end{array}$$

[3] The type correctness of the proof obligation $(\pi(A) = \pi(A'))$ depends on the *prior* proof obligations $\mu(A) \simeq \mu(A')$.

$$
\begin{array}{l}
(\pi(\texttt{natinjection}) = \pi(\texttt{NATinjection})) \\
\quad = \quad (\quad \lambda(f\colon[\texttt{nat}\to\texttt{int}])\colon \qquad (\lambda(g\colon[\texttt{nat}\to\texttt{int}])\colon\forall(i\colon\texttt{nat})\colon \texttt{0} \leq g(i))(f) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad (\forall(i,j\colon\texttt{nat})\colon f(i) = f(j) \;\supset\; i = j) \\
\qquad\quad = \\
\qquad\qquad \lambda(f\colon[\texttt{NAT}\to\texttt{int}])\colon \\
\qquad\qquad\qquad (\lambda(g\colon[\texttt{NAT}\to\texttt{int}])\colon\forall(i\colon\texttt{NAT})\colon g(i) \leq \texttt{0} \;\supset\; g(i) = 0)(f) \;\;) \\
\qquad\qquad\quad \wedge \quad (\forall(i,j\colon\texttt{NAT})\colon f(i) = f(j) \;\supset\; i = j) \\
([\texttt{nat}\to\texttt{int}] \simeq [\texttt{NAT}\to\texttt{int}]) \\
\quad = \quad (\texttt{int} \simeq \texttt{int}); \\
\qquad\quad (\lambda(i\colon\texttt{int})\colon \texttt{0} \leq i) = (\lambda(i\colon\texttt{int})\colon i \leq \texttt{0} \;\supset\; i = 0); (\texttt{int} \simeq \texttt{int})
\end{array}
$$

■

A basic question during typechecking is whether two types are *compatible*, that is, have the same maximal supertype. Two types are said to be compatible if the type equivalence proof obligations on their respective maximal supertypes are provable. The provability of a formula a under context Γ is represented as $\vdash_\Gamma a$.

Definition 3.8 (compatible) Two types A and B are said to be compatible in context Γ (in notation, $(A \sim B)_\Gamma$) if $\vdash_\Gamma a$, for each a in $(\mu(A) \simeq \mu(B))$.[4] ■

We now extend the definition of δ to the case of subtypes so that it leaves the predicate unchanged but expands the definition of the supertype.

Definition 3.9 (expanded type with subtypes)

$$\delta(\Gamma)(\{x\colon T \mid a\}) \quad = \quad \{x\colon \delta(\Gamma)(T) \mid a\}$$

■

We now extend the definition τ to the case of subtypes. Here we could force τ to always return a maximal supertype but this is not done in Definition 3.10 since it would weaken the soundness theorem without significantly simplifying the definition of the type rules. The typechecking of contexts has to be modified to generate a *nonemptiness* proof obligation for the type of any constant declaration. A constant of an empty type would lead to an inconsistent context, and this would mean that constant declarations are not conservative extensions. This modification to Definition 2.4 is not needed for soundness since an inconsistent context makes soundness trivial. It is needed to show

[4]The PVS proof rules are described in Chapter 7.

that constant declarations and definitions are conservative extensions. Note that with subtypes, the type rule for an application is modified to check that the domain type of the function is compatible with the type of its argument, and that the argument satisfies any constraints imposed by the domain type of the function. The case of projection expressions is also not straightforward since the argument type can be a subtype of a tuple type. In this case, we use the direct supertype (see Definition 3.2) which must be a tuple type.

Definition 3.10 (type rules with subtypes)

$$\begin{array}{rcl}
\tau()(\Gamma, c{:}\,T) & = & \texttt{CONTEXT},\ \textit{if}\ \Gamma(c)\ \textit{is undefined}, \\
 & & \tau(\Gamma)(T) = \texttt{TYPE}, \\
 & & \tau()(\Gamma) = \texttt{CONTEXT},\ \textit{and} \\
 & & \vdash_\Gamma (\exists(x{:}\,T){:}\,\texttt{TRUE}) \\
\tau(\Gamma)(\{x{:}\,T \mid a\}) & = & \texttt{TYPE},\ \textit{if}\ \Gamma(x)\ \textit{is undefined}, \\
 & & \tau(\Gamma)(T) = \texttt{TYPE},\ \textit{and}\ \tau(\Gamma, x{:}\,\texttt{VAR}\ T)(a) = \texttt{bool} \\
\tau(\Gamma)(f\ a) & = & B,\ \textit{where}\ \mu_0(\tau(\Gamma)(f)) = [A{\rightarrow}B], \\
 & & \tau(\Gamma)(a) = A', \\
 & & (A \sim A')_\Gamma, \\
 & & \vdash_\Gamma \pi(A)(a) \\
\tau(\Gamma)(\mathtt{p}_i\ a) & = & A_i,\ \textit{where}\ \mu_0(\tau(\Gamma)(a)) = [A_1, A_2]
\end{array}$$

■

Example 3.11 (typechecking subtypes) Let Γ contain the above declarations of `int`, `nat`, `0`, $\leq$, and `natinjection`.

$$\begin{array}{l}
\tau(\Gamma)(\{i{:}\,\texttt{int} \mid 0 \leq i\}) \\
\quad = \quad \texttt{TYPE} \\
\tau(\Gamma)((\lambda(f{:}\,\texttt{natinjection}){:}\,f(0))(\lambda(i{:}\,\texttt{nat}){:}\,i)) \\
\quad = \quad \delta(\Gamma)(\texttt{nat}),\ \text{if} \\
\qquad\quad (\texttt{natinjection} \sim [\texttt{nat}{\rightarrow}\texttt{nat}])_\Gamma, \\
\qquad\quad \vdash_\Gamma \forall(j, k{:}\,\texttt{nat}){:}\,(\lambda(i{:}\,\texttt{nat}){:}\,i)(j) = (\lambda(i{:}\,\texttt{nat}){:}\,i)(k)\ \supset\ j = k, \\
\qquad\quad (\texttt{int} \sim \texttt{nat})_\Gamma,\ \text{and} \\
\qquad\quad \vdash_\Gamma \texttt{0} \leq \texttt{0}
\end{array}$$

■

Only one additional clause to Definition 2.6 is needed to capture the semantics of predicate subtypes.

Definition 3.12 (meaning function with subtypes)

$$\begin{array}{l}\mathcal{M}(\Gamma \mid \gamma)(\{x{:}\, T \mid a\}) \\ \quad = \quad \{y \in \mathcal{M}(\Gamma \mid \gamma)(T) \mid \mathcal{M}(\Gamma, x{:}\, \texttt{VAR}\ T \mid \gamma\{x \leftarrow y\})(a) = \mathbf{1}\}\end{array}$$

■

Example 3.13 (semantics of predicate subtypes) If we assign the usual truth table interpretation to the Boolean function $\supset$:

$$\begin{array}{l}\mathcal{M}(\Gamma \mid \gamma)(\{f{:}\, [\texttt{bool}{\rightarrow}\texttt{bool}] \mid \forall(x{:}\, \texttt{bool}){:}\, x\ \supset\ f(x)\}) \\ \quad = \quad \{\{\langle \mathbf{0}, \mathbf{0}\rangle, \langle \mathbf{1}, \mathbf{1}\rangle\}, \{\langle \mathbf{0}, \mathbf{1}\rangle, \langle \mathbf{1}, \mathbf{1}\rangle\}\}.\end{array}$$

■

The following useful propositions are easily proved from the definitions given above. Proposition 3.14 asserts that the maximal supertype of a type is well typed. Proposition 3.15 asserts that the denotation of a type is a subset of the denotation of its maximal supertype. Proposition 3.16 asserts that if all the proof obligations in $(A \simeq A')$ are valid relative to a given assignment γ for context Γ, then the denotations of A and A' under γ are equal.

Proposition 3.14 *If* $\tau()(\Gamma) = \texttt{CONTEXT}$ *and* $\tau(\Gamma)(A) = \texttt{TYPE}$*, then* $\tau(\Gamma)(\mu(A)) = \texttt{TYPE}$.

Proposition 3.15 *If* $\tau()(\Gamma) = \texttt{CONTEXT}$*,* $\tau(\Gamma)(A) = \texttt{TYPE}$*, and* γ *satisfies* Γ*, then*

1. $\mathcal{M}(\Gamma \mid \gamma)(A) \subseteq \mathcal{M}(\Gamma \mid \gamma)(\mu(A))$ *and*

2. $\mathcal{M}(\Gamma \mid \gamma)(A) \subseteq \mathcal{M}(\Gamma \mid \gamma)(\mu_0(A))$.

Proposition 3.16 *If* A *and* A' *are maximal types in context* Γ*, i.e.,*

1. $\tau()(\Gamma) = \texttt{CONTEXT}$,

2. $\tau(\Gamma)(A) = \tau(\Gamma)(A') = \texttt{TYPE}$,

3. $\mu(A) = A$ *and* $\mu(A') = A'$

and for each a *in* $(A \simeq A')$,

1. $a \equiv \texttt{TRUE}$, *or*

2. $a \equiv (a_1 = a_2)$ *and* $\mathcal{M}(\Gamma \mid \gamma)(a_1) = \mathcal{M}(\Gamma \mid \gamma)(a_2)$ *holds,*

then $\mathcal{M}(\Gamma \mid \gamma)(A) = \mathcal{M}(\Gamma \mid \gamma)(A')$.[5]

Proposition 3.17 *If* $\tau()(\Gamma) = \texttt{CONTEXT}$ *and* $\tau(\Gamma)(T) = \texttt{TYPE}$, *then* $\mathcal{M}(\Gamma \mid \gamma)(T) = \mathcal{M}(\Gamma \mid \gamma)(\{x{:}\,\mu(T) \mid \pi(T)(x)\})$.

We can now examine the updated forms of the invariants given by Theorems 2.11, 2.12, and 2.13. The proof of Theorem 2.11 remains straightforward. The statement of Theorem 2.13 must now be strengthened to include soundness, that is, if $\vdash_\Gamma a$ and γ satisfies Γ, then $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}$. For now, we assume soundness (Theorem 7.2) since we have not yet presented the proof rules.

Theorem 3.18 (type soundness) *If* $\tau()(\Gamma) = \texttt{CONTEXT}$, γ *satisfies* Γ, *and* $\tau(\Gamma)(A) = \texttt{TYPE}$ *then* $\mathcal{M}(\Gamma \mid \gamma)(A) \in U$.

Proof. There is only one new case to add to the induction proof of Theorem 2.12, namely, when $A \equiv \{x{:}\,T \mid a\}$. In this case, by Definition 3.10, $\tau(\Gamma)(T) = \texttt{TYPE}$, so by the induction hypothesis, $\mathcal{M}(\Gamma \mid \gamma)(T) \in U$. Since, by Definition 3.12, $\mathcal{M}(\Gamma \mid \gamma)(A) \subseteq \mathcal{M}(\Gamma \mid \gamma)(T)$, we have $\mathcal{M}(\Gamma \mid \gamma)(A) \in U$ by Definition 1.1. ■

Theorem 3.19 (term soundness) *If* $\tau()(\Gamma) = \texttt{CONTEXT}$, γ *satisfies* Γ, *and* $\tau(\Gamma)(a) = A$ *then* $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$.

Proof. There are two affected cases in the proof from that of Theorem 2.13, namely, those of application and projection. The case of projection expressions is straightforward given Proposition 3.15.

When $a \equiv (f\ b)$, by Definition 3.10, we have that $\tau(\Gamma)(f) = [B \rightarrow A]$ and $\tau(\Gamma)(b) = B'$. Let X be $\mathcal{M}(\Gamma \mid \gamma)(B)$, X' be $\mathcal{M}(\Gamma \mid \gamma)(B')$, and Y be $\mathcal{M}(\Gamma \mid \gamma)(A)$. Then by Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)([B \rightarrow A]) = Y^X$. By the induction hypotheses, $\mathcal{M}(\Gamma \mid \gamma)(f) \in Y^X$ and $\mathcal{M}(\Gamma \mid \gamma)(b) \in X'$. By Definition 3.10, soundness of the proof rules (Theorem 7.2), and Propositions 3.15 and 3.16, there is a maximal supertype $\mu(B)$ of both B and B' such that X and X' are both subsets of $\mathcal{M}(\Gamma \mid \gamma)(\mu(B))$. Since, by Definition 3.10, $\vdash_\Gamma \pi(B)(b)$, and by Proposition 3.17, $\mathcal{M}(\Gamma \mid \gamma)(B) = \mathcal{M}(\Gamma \mid \gamma)(\{x{:}\,\mu(B) \mid \pi(B)(x)\})$, we have $\mathcal{M}(\Gamma \mid \gamma)(b) \in \mathcal{M}(\Gamma \mid \gamma)(B)$, and hence by Definition 2.6, $\mathcal{M}(\Gamma \mid \gamma)((f\ b)) \in \mathcal{M}(\Gamma \mid \gamma)(A)$. ■

[5]We remind the reader that the formulas a in $(A \simeq A')$ are equalities, but we have not yet formally introduced equality into the language.

3.1 Summary

PVS features a form of subtyping where it is possible to form the subtype of a type satisfying a given predicate on the type. This kind of subtyping introduces several delicate semantic issues into PVS. A term can now have several types since, for example, the term corresponding to the number 2 can be a prime number, an even number, a natural number, an integer, a rational number, or a real number. When the expected type is a subtype, the canonical type of the actual term must be compatible with the expected type, that is, the two maximal supertypes must be equivalent and the actual term must satisfy any subtype constraints imposed by the expected type. We have defined the notions of maximal supertype, subtype constraints, type equivalence, and compatibility. These notions are used to define the type rules and semantics of the simply typed fragment of PVS extended with subtypes. Note that both type equivalence (and hence, compatibility) and type correctness are undecidable. Proof obligations generated during typechecking are the only source of such undecidability. The modularization of the type system into a decidable part consisting of the simply typed fragment, and the proof obligations generated by subtyping, is perhaps the most significant design consideration in the PVS language.

Chapter 4

Dependent Types

The PVS language fragment described thus far is already quite expressive. It employs definitional equivalence between types and contains predicate subtypes. It is undecidable whether an expression in this fragment is type-correct because of the proof obligations that arise with respect to predicate subtypes and type equivalence. The next step is the addition of type dependencies between the components of a type. This extension considerably enhances the utility of this type system. It is also a natural extension given predicate subtyping which already allows types that *depend* on free variables in the predicates. With dependent typing, we can make the type of one component of a product depend on the value of another component, or the type of the range of a function vary according to its argument value.

A dependent product type is written as $[x\colon A, B]$. A dependent function type is written as $[x\colon A \rightarrow B]$. Any product or function type can be transformed into a dependent type by inserting dummy type bindings. Conversely, any dummy type bindings that do not actually bind any variable occurrences can be removed. The type rules and semantics below will assume that all product and function types are presented as dependent types.

Example 4.1 (dependent types)

$$\begin{array}{l}[i\colon \texttt{nat}, \{j\colon \texttt{nat} \mid j \leq i\}],\\ [i\colon \texttt{nat}, [\{j\colon \texttt{nat} \mid j \leq i\} \rightarrow \texttt{bool}]],\\ [i\colon \texttt{int} \rightarrow \{j\colon \texttt{int} \mid i \leq j\}].\end{array}$$

■

Before we treat dependent types, we update the definitions of the set of free variables and substitution to account for the fact that with subtyping and dependent typing, both free and bound variables can occur in terms and types. This is needed for the next step where we try to remove type dependencies by substituting a term into a dependent type.

Definition 4.2 (free variables for types)

$$\begin{array}{rcl} FV(\Gamma)([x\colon A \to B]) & = & FV(\Gamma)(A) \cup (FV(\Gamma, x\colon \mathtt{VAR}\ A)(B) - \{x\}) \\ FV(\Gamma)([x\colon A, B]) & = & FV(\Gamma)(A) \cup (FV(\Gamma, x\colon \mathtt{VAR}\ A)(B) - \{x\}) \\ FV(\Gamma)(\{x\colon A \mid a\}) & = & FV(\Gamma)(A) \cup (FV(\Gamma, x\colon \mathtt{VAR}\ A)(a) - \{x\}) \end{array}$$

■

Definition 4.3 (substitution for types)

$$\begin{array}{l} [x\colon A \to B][a_1/x_1, \ldots, a_n/x_n] \\ \quad = \quad [y\colon A[a_1/x_1, \ldots, a_n/x_n] \to B[y/x, a_1/x_n, \ldots, a_n/x_n]] \\ [x\colon A, B][a_1/x_1, \ldots, a_n/x_n] \\ \quad = \quad [y\colon A[a_1/x_1, \ldots, a_n/x_n], B[y/x, a_1/x_1, \ldots, a_n/x_n]] \\ \{x\colon A \mid a\}[a_1/x_n, \ldots, a_n/x_n] \\ \quad = \quad \{y\colon A[a_1/x_1, \ldots, a_n/x_n] \mid a[y/x, a_1/x_n, \ldots, a_n/x_n]\} \end{array}$$

where y is a fresh variable. ■

The definition of μ has to be modified slightly for dependent types. The definition is first extended to type bindings, $\mu(x\colon T) = x\colon \mu(T)$. The definition for the case of dependent function types is unchanged so that $\mu([x\colon A \to B]) = [x\colon A \to \mu(B)]$. The definition for the product case is more delicate since the definition $\mu([x\colon A, B]) = [x\colon \mu(A), \mu(B)]$ results in a loss of type information regarding the occurrences of x in B.[1] To ensure that type information regarding x is retained, we define a new operation $T \backslash a$ which constrains the subtype assertions in type T with an additional assertion a.

Definition 4.4 (Adding subtype constraints)

$$\begin{array}{rcl} s \backslash a & = & s \\ \{x\colon T | b\} \backslash a & = & \{x\colon T | a \wedge b\} \\ [A \to B] \backslash a & = & [A \backslash a \to B \backslash a] \\ [A, B] \backslash a & = & [A \backslash a, B \backslash a] \end{array}$$

■

[1]Doug Howe brought this problem to our attention.

We can now define the maximal supertype operation for dependent tuple types.

Definition 4.5 (Maximal supertype for dependent product types)

$$\mu([x\colon A, B]) \;=\; [x\colon \mu(A), B\backslash\pi(A)(x)]$$

■

The definition of π for a dependent function type $[y\colon A\to B]$ is slightly different from that of an ordinary function type since $\pi(B)$ can contain free occurrences of the variable y. For example, $\pi([i\colon \texttt{int}\to\{j\colon \texttt{int} \mid i \leq j\}])$ must be $\lambda(f\colon [i\colon \texttt{int}\to\texttt{int}])\colon (\forall(i\colon \texttt{int})\colon i \leq f(i))$. The definition for dependent tuples remains essentially unchanged from that of ordinary products.

Definition 4.6 (constraint predicates for dependent types)

$$\begin{array}{rcl}\pi([y\colon A\to B]) & = & (\lambda(x\colon [y\colon A\to\mu(B)])\colon (\forall(y\colon A)\colon \pi(B)(x(y))))\\ \pi([y\colon A, B]) & = & (\lambda(x\colon [y\colon \mu(A), \mu(B)\backslash\pi(A)(y)])\colon\\ & & \quad \pi(A)(\mathtt{p}_1\ x) \wedge \pi(B)(\mathtt{p}_2\ x)[(\mathtt{p}_1\ x)/y])\end{array}$$

■

Example 4.7 (dependent type predicates)

$$\begin{array}{rcl}\mu([i\colon \texttt{int}\to\{j\colon \texttt{int} \mid i \leq j\}]) & = & [i\colon \texttt{int}\to\texttt{int}]\\ \pi([i\colon \texttt{int}\to\{j\colon \texttt{int} \mid i \leq j\}]) & = & \lambda(f\colon [i\colon \texttt{int}\to\texttt{int}])\colon\\ & & \quad \forall(i\colon \texttt{int})\colon (\lambda(j\colon \texttt{int})\colon i \leq j)(f(i))\end{array}$$

■

The definition of $\simeq$ must also be massaged slightly for dependent types. Recall that $\simeq$ checks whether two maximal types are equivalent by generating proof obligations as needed. This is the basic operation for checking whether the expected type of an expression is compatible with its actual type. The subtlety now is that the expected type might be a dependent type where the actual type is not. Consider the case of the pair $\langle 5, (\lambda(x\colon \{j\colon \texttt{nat} \mid j \leq 5\})\colon x)\rangle$ whose type would be computed by τ as $[i\colon \texttt{nat}, [\{j\colon \texttt{nat} \mid j \leq 5\}\to\{j\colon \texttt{nat} \mid j \leq 5\}]]$ where the expected type might be $[i\colon \texttt{nat}, [\{j\colon \texttt{nat} \mid j \leq i\}\to\{j\colon \texttt{nat} \mid j \leq i\}]]$. To cope with this, we will allow the option of two maximal types, say A and B, to be compared using $\simeq$ in the context of an expression a. This is indicated by the notation $(A \simeq B)/a$. Note that $(A \simeq B)/a$ is sensible only when A and B are maximal types. The missing cases in Definition 3.6 are included in Definition 4.8. For a list of formulas $a_1, \ldots, a_n$, let $(\forall (x{:}\, T){:}\, a_1, \ldots, a_n)$ represent the list $(\forall (x{:}\, T){:}\, a_1), \ldots, (\forall (x{:}\, T){:}\, a_n)$.[2]

Definition 4.8 (type equivalence for dependent types)

$$
\begin{array}{rcl}
(s \simeq s)/a & = & \mathtt{TRUE} \\
([x{:}\, A \to B] \simeq [x'{:}\, A' \to B']) & = & (\mu(A) \simeq \mu(A')); \\
 & & (\pi(A) = \pi(A')); \\
 & & (\forall (x{:}\, A){:}\, (B \simeq B'[x/x'])) \\
([x{:}\, A \to B] \simeq [x'{:}\, A' \to B'])/a & = & (\mu(A) \simeq \mu(A')); \\
 & & (\pi(A) = \pi(A')); \\
 & & (\forall (x{:}\, A){:}\, (B \simeq B'[x/x'])/a(x)) \\
([x{:}\, A_1, A_2] \simeq [y{:}\, B_1, B_2]) & = & (A_1 \simeq B_1); \\
 & & (\forall (x{:}\, A_1){:}\, (A_2 \simeq B_2[x/y])) \\
([x{:}\, A_1, A_2] \simeq [y{:}\, B_1, B_2])/a & = & (A_1 \simeq B_1)/(\mathtt{p}_1\ a); \\
 & & (A_2[(\mathtt{p}_1\ a)/x] \simeq B_2[(\mathtt{p}_1\ a)/y])/(\mathtt{p}_2\ a) \\
(A \simeq B)/a & = & \mathtt{FALSE},\ \textit{otherwise.}
\end{array}
$$

■

As with $(A \sim B)_\Gamma$, the notation $(A \overset{a}{\sim} B)_\Gamma$ indicates that all the proof obligations a' in $(\mu(A) \simeq \mu(B))/a$ are provable, that is, $\vdash_\Gamma a'$.

With dependent types, the type rules must be modified so as to augment the context suitably to account for any dependencies. We will give the definitions only for dependent type constructions.

Definition 4.9 (type rules with dependent types)

$$
\begin{array}{rcl}
\tau(\Gamma)([x{:}\, A, B]) & = & \mathtt{TYPE},\ \textit{if}\ \Gamma(x)\ \textit{is undefined}, \\
 & & \tau(\Gamma)(A) = \mathtt{TYPE},\ \textit{and} \\
 & & \tau(\Gamma, x{:}\, \mathtt{VAR}\ A)(B) = \mathtt{TYPE} \\
\tau(\Gamma)([x{:}\, A \to B]) & = & \mathtt{TYPE},\ \textit{if}\ \Gamma(x)\ \textit{is undefined}, \\
 & & \tau(\Gamma)(A) = \mathtt{TYPE},\ \textit{and} \\
 & & \tau(\Gamma, x{:}\, \mathtt{VAR}\ A)(B) = \mathtt{TYPE}
\end{array}
$$

[2]Note that the type-correctness of the proof obligation $(\pi(A) = \pi(A'))$ in Definition 4.8 depends on the *prior* proof obligations $\mu(A) \simeq \mu(A')$.

$$\begin{array}{rcl}
\tau(\Gamma)(f\ a) & = & B',\ \mathit{where}\ \mu_0(\tau(\Gamma)(f)) = [x\colon A \to B], \\
 & & \tau(\Gamma)(a) = A', \\
 & & (A \overset{a}{\sim} A')_\Gamma, \\
 & & B'\ \mathit{is}\ B[a/x], \\
 & & \vdash_\Gamma \pi(A)(a) \\
\tau(\Gamma)(\lambda(x\colon A)\colon a) & = & [x\colon A \to B],\ \mathit{where} \\
 & & B = \tau(\Gamma, x\colon \mathtt{VAR}\ A)(a) \\
\tau(\Gamma)(\mathtt{p}_1\ a) & = & A_1,\ \mathit{where}\ \mu_0(\tau(\Gamma)(a)) = [x\colon A_1, A_2] \\
\tau(\Gamma)(\mathtt{p}_2\ a) & = & A_2[(\mathtt{p}_1\ a)/x],\ \mathit{where}\ \mu_0(\tau(\Gamma)(a)) = [x\colon A_1, A_2]
\end{array}$$

■

Example 4.10 (dependent typing)

$$\begin{array}{rcl}
\tau(\Gamma)([x\colon \mathtt{bool}, \{y\colon \mathtt{bool} \mid x \supset y\}]) & = & \mathtt{TYPE} \\
\tau(\Gamma)([x\colon \mathtt{bool} \to \{y\colon \mathtt{bool} \mid x \supset y\}]) & = & \mathtt{TYPE}
\end{array}$$

■

Before we can assign meanings to dependent types, we must augment our definition of the universe U to contain sets corresponding to these constructions. If F is a function with domain set X and a range Y, which is a set of sets, we can define ΣF to be the set $\{\langle x, y\rangle \mid x \in \mathit{dom}(F), y \in F(x)\}$ and ΠF to be the set $\{f \mid (\forall x \in \mathit{dom}(F)\colon f(x) \in F(x))\}$. Note that $\Pi F \subseteq \bigcup_{X \in \Sigma F} \wp(X)$ but we include ΠF in the universe U defined below for simplicity. We can drop $X \times Y$ and X^Y from the universe definition since $X \times Y$ can be obtained from ΣF by defining an F with domain X that always returns Y, and similarly, X^Y can be obtained by ΠF where F is defined to with domain Y to always return X. The universe U can then be redefined as below.

Definition 4.11 (type universe with dependent types)

$$\begin{array}{rcl}
U_0 & = & \{\mathbf{2}, \mathbf{R}\} \\
U_{i+1} & = & U_i \\
 & & \cup \bigcup_{X \in U_i} \wp(X) \\
 & & \cup \{\Sigma F \mid F \in W_i\}
\end{array}$$

$$
\begin{aligned}
&\cup \{\Pi F \mid F \in W_i\} \\
W_i &= \bigcup_{X \in U_i} U_i^X \\
U_\omega &= \bigcup_{i \in \omega} U_i \\
U &= U_\omega
\end{aligned}
$$

■

One very important consequence of the above extension of the universe is that all type dependencies must be bounded in the sense that if B is a type expression with a single free variable x of type A, then it must be the case that for any set $[\![A]\!]$ representing A, there is a bound n such that for any z in $[\![A]\!]$, the meaning of B under $\{x \leftarrow z\}$ must be in U_n. This property is easily proved by induction on the structure of a PVS type since the parameter x can appear only in the predicate part of a subtype where the rank of the meaning of the resulting type cannot vary with the value of x. In particular, there is no way to define a type constructor T^n in PVS that returns the n-tuple $\underbrace{[T, [\ldots, T]]}_{n}$ for a given n since this would entail an unbounded dependency. If unbounded type dependencies were allowed in PVS, one can construct a dependent type such as $[n\colon nat \rightarrow T^n]$ whose representation is not in U as defined above.

The meaning function for dependent types is obtained by adding the cases corresponding to dependent product and function types. All the other cases are unchanged from Definition 3.12. Note that the semantic definition for dependent types is equivalent to the nondependent one when there are no dependencies.

Definition 4.12 (meaning function with dependent types)

$$
\begin{aligned}
\mathcal{M}(\Gamma \mid \gamma)([x\colon A, B]) &= \Sigma F,\ \textit{where} \\
&\quad F\ \textit{maps}\ z \in \mathcal{M}(\Gamma \mid \gamma)(A)\ \textit{to} \\
&\quad \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow z\})(B) \\
\mathcal{M}(\Gamma \mid \gamma)([x\colon A \rightarrow B]) &= \Pi F,\ \textit{where} \\
&\quad F\ \textit{maps}\ z \in \mathcal{M}(\Gamma \mid \gamma)(A)\ \textit{to} \\
&\quad \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow z\})(B)
\end{aligned}
$$

■

Example 4.13 (meaning function with dependent types)

$$\begin{aligned}\mathcal{M}(\Gamma \mid \gamma)([x{:}\,\texttt{bool}, \{y{:}\,\texttt{bool} \mid x \supset y\}]) &= \{\langle \mathbf{0},\mathbf{0}\rangle, \langle \mathbf{0},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{1}\rangle\} \\ \mathcal{M}(\Gamma \mid \gamma)([x{:}\,\texttt{bool}{\rightarrow}\{y{:}\,\texttt{bool} \mid x \supset y\}]) &= \{\{\langle \mathbf{0},\mathbf{0}\rangle, \langle \mathbf{1},\mathbf{1}\rangle\}, \\ &\quad\ \{\langle \mathbf{0},\mathbf{1}\rangle, \langle \mathbf{1},\mathbf{1}\rangle\}\}\end{aligned}$$

■

We now need to show that the extensions corresponding to dependent types preserve the properties in Theorems 3.18 and 3.19, namely, $\mathcal{M}(\Gamma \mid \gamma)(T) \in U$ and $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(\tau(\Gamma)(a))$. For the former, we prove a stronger theorem that incorporates the rank-boundedness of dependent types.

Theorem 4.14 (rank bounded type semantics) *If B is a pretype, $x_1, \ldots, x_n$ is a list of symbols, $A_1, \ldots, A_n$ is a list of pretypes such that*

1. $\tau()(\Gamma, x_1{:}\,\texttt{VAR}\ A_1, \ldots, x_n{:}\,\texttt{VAR}\ A_n) = \texttt{CONTEXT}$,

2. $\tau(\Gamma, x_1{:}\,\texttt{VAR}\ A_1, \ldots, x_n{:}\,\texttt{VAR}\ A_n)(B) = \texttt{TYPE}$, *and*

3. *γ is an assignment satisfying Γ,*

then there is an i such that for any list of values $z_1, \ldots, z_n$ where $\gamma\{x_1 \leftarrow z_1\} \ldots \{x_n \leftarrow z_n\}$ is a satisfying assignment for $\Gamma, x_1{:}\,\texttt{VAR}\ A_1, \ldots, x_n{:}\,\texttt{VAR}\ A_n$, we have

$$\mathcal{M}(\Gamma, x_1{:}\,\texttt{VAR}\ A_1, \ldots, x_n{:}\,\texttt{VAR}\ A_n \mid \gamma\{x_1 \leftarrow z_1\} \ldots \{x_n \leftarrow z_n\})(B) \in U_i.$$

Proof. The proof is by structural induction on the pretype B. Let Γ' denote $\Gamma, x_1{:}\,\texttt{VAR}\ A_1, \ldots, x_n{:}\,\texttt{VAR}\ A_n$, γ' denote $\gamma\{x_1 \leftarrow z_1\} \ldots \{x_n \leftarrow z_n\}$, and $[\![C]\!]$ denote $\mathcal{M}(\Gamma' \mid \gamma')(C)$.

1. $B \equiv s$: Since $[\![B]\!]$ is just $\gamma(B)$ by Definition 2.6, we have that there is an i such that $[\![B]\!] \in U_i$ regardless of the choice of values $z_1, \ldots, z_n$.

2. $B \equiv \{y{:}\,T \mid a\}$: By the induction hypothesis, we know that for some j, it is always the case that $[\![T]\!] \in U_j$. By Definition 3.12, we have that $[\![B]\!] \subseteq [\![T]\!]$ so if we let $i = j + 1$, then by Definition 4.11, it is always the case that $[\![B]\!] \in U_i$.

3. $B \equiv [y\colon C \rightarrow D]$: By Definition 4.9, $\Gamma'(y)$ is undefined, $\tau(\Gamma')(C) = \texttt{TYPE}$, $\tau()(\Gamma', y\colon \texttt{VAR}\ C) = \texttt{CONTEXT}$, and $\tau(\Gamma', y\colon \texttt{VAR}\ C)(D) = \texttt{TYPE}$. By the induction hypothesis, for some j, it is always the case that $\mathcal{M}(\Gamma' \mid \gamma')(C) \in U_j$, and for some k, it is always the case that for any satisfying assignment $\gamma'\{y \leftarrow w\}$ for $\Gamma', y\colon \texttt{VAR}\ C$, we have $\mathcal{M}(\Gamma', y\colon \texttt{VAR}\ C \mid \gamma'\{y \leftarrow w\})(D) \in U_k$. Then the function F mapping w in $\mathcal{M}(\Gamma')(C)$ to $\mathcal{M}(\Gamma', y\colon \texttt{VAR}\ C \mid \gamma'\{y \leftarrow w\})(D)$ is an element of W_{j+k}. Letting i be $j + k + 1$, we have by Definition 4.12 that $\mathcal{M}(\Gamma' \mid \gamma')(B)$ is ΠF and is hence an element of U_i by Definition 4.11.

4. $B \equiv [y\colon C, D]$: Similar to the previous case.

■

By choosing n to be 0, the previous theorem yields the result that when $\tau(\Gamma)(B) = \texttt{TYPE}$, $\mathcal{M}(\Gamma \mid \gamma)(B) \in U$.

We next need to establish that for any preterm a, if $\tau(\Gamma)(a) = A$, then $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$. The first step in this direction is the proof of the substitution lemma below.

Proposition 4.15 *If $\tau()(\Gamma) = \tau()(\Gamma') = \texttt{CONTEXT}$ where for each s, $\Gamma(s)$ is defined if and only if $\Gamma'(s)$ is defined, and γ is an assignment satisfying both Γ and Γ', then*

1. *If $\Gamma(s) = \Gamma'(s)$ (i.e., they are equal when either $\Gamma(s)$ or $\Gamma'(s)$ is defined), then*

   (a) $\tau(\Gamma)(a) = \tau(\Gamma')(a)$, for any preterm a.

   (b) $\tau(\Gamma)(A) = \tau(\Gamma')(A)$, for any pretype A.

2. *$\mathcal{M}(\Gamma \mid \gamma)(A) = \mathcal{M}(\Gamma' \mid \gamma)(A)$, when $\tau(\Gamma)(A) = \texttt{TYPE}$.*

3. *$\mathcal{M}(\Gamma \mid \gamma)(a) = \mathcal{M}(\Gamma' \mid \gamma)(a)$, for any preterm a such that $\tau(\Gamma)(a)$ is defined.*

Lemma 4.16 (substitution lemma) *If $\tau()(\Gamma, x\colon \texttt{VAR}\ A) = \texttt{CONTEXT}$, $\tau(\Gamma)(a) = A$, then*

1. *If $\tau(\Gamma, x\colon \texttt{VAR}\ A)(b) = B$, then*
   $\mathcal{M}(\Gamma \mid \gamma)(b[a/x]) = \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(b)$.

2. *If $\tau(\Gamma, x\colon \texttt{VAR}\ A)(C) = \texttt{TYPE}$, then*
   $\mathcal{M}(\Gamma \mid \gamma)(C[a/x]) = \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(C)$.

Proof. The proof is by simultaneous structural induction on the preterm b and the pretype C. The following cases deal with the preterm b.

1. $b \equiv s$: If $s \equiv x$, then by Definition 4.12, the left-hand side $\mathcal{M}(\Gamma \mid \gamma)(b[a/x])$ is $\mathcal{M}(\Gamma \mid \gamma)(a)$, and the right-hand side $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(b)$ is also $\mathcal{M}(\Gamma \mid \gamma)(a)$.

   If $s \not\equiv x$, then by Definition 4.12, the left-hand side and the right-hand side are both equal to $\gamma(s)$.

2. $b \equiv (\lambda(y{:}\,C){:}\,d)$: Since C can contain free occurrences of x, we have by the induction hypothesis that $\mathcal{M}(\Gamma \mid \gamma)(C[a/x]) = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(C)$. Also, $\mathcal{M}(\Gamma \mid \gamma)((\lambda(y{:}\,C){:}\,d)[a/x])$ is equal to the set of ordered pairs $\langle v, z\rangle$ such that $v \in \mathcal{M}(\Gamma \mid \gamma)(C[a/x])$ and $z = \mathcal{M}(\Gamma, y{:}\,\texttt{VAR}\ C[a/x] \mid \gamma\{y \leftarrow v\})(d[a/x])$.

   By the induction hypothesis, $\mathcal{M}(\Gamma, y{:}\,\texttt{VAR}\ C[a/x] \mid \gamma\{y \leftarrow v\})(d[a/x]) = \mathcal{M}(\Gamma, y{:}\,\texttt{VAR}\ C[a/x], x{:}\,\texttt{VAR}\ A \mid \gamma\{y \leftarrow v\}\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(d)$. Since x does not occur free in $C[a/x]$, by Proposition 4.15 we can exchange the occurrences of y and x so that $\mathcal{M}(\Gamma, y{:}\,\texttt{VAR}\ C[a/x], x{:}\,\texttt{VAR}\ A \mid \gamma\{y \leftarrow v\}\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(d) = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A, y{:}\,\texttt{VAR}\ C[a/x] \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\}\{y \leftarrow v\})(d)$.

   By Definition 4.12, the right-hand side is the set of ordered pairs of the form $\langle v, z\rangle$ such that $v \in \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(C)$ and $z = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A, y{:}\,\texttt{VAR}\ C \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\}\{y \leftarrow v\})(d)$. By Proposition 4.15 and the induction hypothesis, we know that $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A, y{:}\,\texttt{VAR}\ C \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\}\{y \leftarrow v\})(d) = \mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A, y{:}\,\texttt{VAR}\ C[a/x] \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\}\{y \leftarrow v\})(d)$, and hence it follows that the two sets of ordered pairs are equal.

3. $b \equiv (f\ \ c)$: In this case, $b[a/x] \equiv (f[a/x]\ \ c[a/x])$ and the conclusion follows easily from the induction hypothesis and Definition 4.12.

4. $b \equiv (b_1, b_2)$: The conclusion follows easily from Definitions 2.15, 4.12, and the induction hypotheses.

5. $b \equiv (\texttt{p}_i\ \ c)$: This case is also straightforward since $b[a/x] \equiv (\texttt{p}_i\ \ c[a/x])$, and by the induction hypothesis, $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ a \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(c) = \mathcal{M}(\Gamma \mid \gamma)(c[a/x])$.

The remaining cases deal with the pretype C.

1. $C \equiv s$: This case is trivial since by Definition 2.15, $C[a/x] \equiv C$ and the left-hand and right-hand sides both reduce to $\gamma(C)$.

2. $C \equiv \{y\colon T \mid d\}$: The argument here follows along the lines of the $b \equiv (\lambda(x\colon C)\colon D)$ case above. By the induction hypotheses, we know that

$$\begin{array}{l} \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(T) \\ \quad = \ \mathcal{M}(\Gamma \mid \gamma)(T[a/x]) \\ \mathcal{M}(\Gamma, y\colon \texttt{VAR}\ T[a/x], x\colon \texttt{VAR}\ A \mid \gamma\{y \leftarrow z\}\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(d) \\ \quad = \ \mathcal{M}(\Gamma, y\colon \texttt{VAR}\ T[a/x] \mid \gamma\{y \leftarrow z\})(d[a/x]), \\ \qquad\quad \text{for any } z \in \mathcal{M}(\Gamma \mid \gamma)(T[a/x]) \end{array}$$

   The conclusion follows from Proposition 4.15 and Definition 4.12.

3. $C \equiv [y\colon C_1 \rightarrow C_2]$: The argument here is similar to that of the previous case. Essentially, by the induction hypothesis and Proposition 4.15, the function mapping $z \in \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ A \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a))(C_1)$ to $\mathcal{M}(\Gamma, y\colon \texttt{VAR}\ C_1[a/x], x\colon \texttt{VAR}\ A \mid \gamma\{y \leftarrow z\}\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(a)\})(C_2)$ is the same as the function mapping $z \in \mathcal{M}(\Gamma \mid \gamma)(C_1[a/x])$ to $\mathcal{M}(\Gamma, y\colon \texttt{VAR}\ C_1[a/x] \mid \gamma\{y \leftarrow z\})(C_2[a/x])$.

4. $C \equiv [y\colon C_1, C_2]$: Similar to the previous case.

■

Proposition 4.17 is stated below without proof. It asserts the semantic equivalence with respect to term a of types A and B when $(A \overset{a}{\sim} B)_\Gamma$ holds. Note that its correctness depends on the soundness of the proof rules.

Proposition 4.17 *If $\tau()(\Gamma) = \texttt{CONTEXT}$, a is a preterm such that $\tau(\Gamma)(a) = B$, and $(A \overset{a}{\sim} B)_\Gamma$, then $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$ iff $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(B)$.*

Theorem 4.18 *If $\tau()(\Gamma) = \texttt{CONTEXT}$, γ is an assignment satisfying Γ, and a is a preterm such that $\tau(\Gamma)(a) = A$, then $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$.*

Proof. The proof is by induction on the structure of the preterm a.

1. $a \equiv s$: Then by Definition 4.12, $\mathcal{M}(\Gamma \mid \gamma)(a) = \gamma(a)$, and by Definition 2.8, we have that $\gamma(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$.

2. $a \equiv (\lambda(x\colon C)\colon b)$: By Definition 4.9, we have $\tau(\Gamma)(a) = A = [x\colon C{\rightarrow}\tau(\Gamma, x\colon \texttt{VAR}\ C)(b)]$. Let B label $\tau(\Gamma, x\colon \texttt{VAR}\ C)(b)$. We know that $\mathcal{M}(\Gamma \mid \gamma)(A)$ is of the form ΠF where F maps $z \in \mathcal{M}(\Gamma \mid \gamma)(C)$ to $\mathcal{M}(\Gamma, x\colon \texttt{VAR}\ C \mid \gamma\{x \leftarrow z\})(B)$.

   By the induction hypothesis on b, we know that for any $z \in \mathcal{M}(\Gamma \mid \gamma)(C)$, $\mathcal{M}(\Gamma, x\colon \texttt{VAR}\ C \mid \gamma\{x \leftarrow z\})(b) \in \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ C \mid \gamma\{x \leftarrow z\})(B)$. Since by Definition 4.12, $\mathcal{M}(\Gamma \mid \gamma)(a)$ is a function mapping $z \in \mathcal{M}(\Gamma \mid \gamma)(C)$ to $\mathcal{M}(\Gamma, x\colon \texttt{VAR}\ C \mid \gamma\{x \leftarrow z\})(b)$, we have $\mathcal{M}(\Gamma \mid \gamma)(a) \in \Pi F$ by the definition of Π.

3. $a \equiv (f\ b)$: By Definition 4.9, we have that $\tau(\Gamma)(f) = [x\colon B{\rightarrow}A']$, $\tau(\Gamma)(b) = B'$, $(B \overset{a}{\sim} B')_\Gamma$, $A \equiv A'[a/x]$, and $\vdash_\Gamma \pi(B)(b)$. We know by the induction hypothesis that $\mathcal{M}(\Gamma \mid \gamma)(f) \in \mathcal{M}(\Gamma \mid \gamma)([x\colon B{\rightarrow}A'])$ and $\mathcal{M}(\Gamma \mid \gamma)(b) \in \mathcal{M}(\Gamma \mid \gamma)(B')$. By Propositions 4.17 and 3.17, $\mathcal{M}(\Gamma \mid \gamma)(b) \in \mathcal{M}(\Gamma \mid \gamma)(\mu(B))$. We therefore have by Proposition 3.17 that $\mathcal{M}(\Gamma \mid \gamma)(b) \in \mathcal{M}(\Gamma \mid \gamma)(B)$. By Definition 4.12, $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma, x\colon \texttt{VAR}\ B \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(b)\})(A')$, and hence by Lemma 4.16 it follows that $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A'[b/x])$.

4. $a \equiv (a_1, a_2)$: The conclusion follows easily from the induction hypothesis and Definition 4.9.

5. $a \equiv (\texttt{p}_i\ b)$: The conclusion follows easily from Proposition 3.17, the induction hypothesis, and Definition 4.9. The $(\texttt{p}_2\ b)$ case also employs Lemma 4.16.

■

4.1 Summary

Dependent typing is a significant enhancement to PVS since it adds an important degree of flexibility and precision to the type system. Notions such as subtype constraints and type equivalence that were introduced for subtyping can be extended for the case of dependent types. The semantic universe must be extended to include additional sets to accommodate the semantics of dependent types. The rank-boundedness of type dependencies is crucial in demonstrating that dependent types can be interpreted in this extended semantic universe.

Chapter 5

Theories and Parametric Theories

The next extension of the PVS language introduces theories and parametric theories. The theory construct of PVS provides a way of packaging together a related collection of declarations. Theories can be parametric in individual or type parameters. Thus, PVS permits polymorphism or type parametricity only at the theory level rather than at the declaration level as in HOL [GM93]. We first consider PVS theories without parameters. The main change now is that contexts are no longer simple and can contain theory declarations as well. A theory declaration has the form $m\colon \texttt{THEORY} = \Delta$, where Δ is a *simple context* with no variable or theory declarations. If $\Gamma(m)$ is the declaration $m\colon \texttt{THEORY} = \Delta$, then $\mathit{kind}(\Gamma(m)) = \texttt{THEORY}$, and $\mathit{definition}(\Gamma(m)) = \Delta$. Correspondingly, constants and type names are no longer just symbols but can be compound names of the form $m.s$ where m is a symbol naming a theory and s is a symbol corresponding to the constant or type name.

5.1 Theories without Parameters

To define the type rules for theories, we first modify the definition of τ for simple contexts so that the context argument is not always empty. Here $\Delta; \Gamma$ represents the concatenation of contexts.

Definition 5.1 (type rules for contexts)

$$\begin{array}{rcl}
\tau(\Theta)(\{\}) & = & \texttt{CONTEXT} \\
\tau(\Theta)(\Gamma, s : \texttt{TYPE} = T) & = & \texttt{CONTEXT},\ \textit{if}\ \Gamma(s)\ \textit{and}\ \Theta(s)\ \textit{are undefined,} \\
 & & \tau(\Theta)(\Gamma) = \texttt{CONTEXT},\ \textit{and}
\end{array}$$

$$\begin{array}{rcl}
 & & \tau(\Theta;\Gamma)(T) = \mathtt{TYPE} \\
\tau(\Theta)(\Gamma, c{:}\,T) & = & \mathtt{CONTEXT},\ \textit{if}\ \Gamma(c)\ \textit{and}\ \Theta(c)\ \textit{are undefined,} \\
 & & \tau(\Theta)(\Gamma) = \mathtt{CONTEXT},\ \textit{and} \\
 & & \tau(\Theta;\Gamma)(T) = \mathtt{TYPE} \\
\tau(\Theta)(\Gamma, x{:}\,\mathtt{VAR}\ T) & = & \mathtt{CONTEXT},\ \textit{if}\ \Gamma(x)\ \textit{and}\ \Theta(x)\ \textit{are undefined,} \\
 & & \tau(\Theta)(\Gamma) = \mathtt{CONTEXT},\ \textit{and} \\
 & & \tau(\Theta;\Gamma)(T) = \mathtt{TYPE}
\end{array}$$

■

Example 5.2 (type rules for contexts)

$$\tau(\Omega)(\mathtt{real}{:}\,\mathtt{TYPE}, \mathtt{0}{:}\,\mathtt{real}, \leq{:}\,[[\mathtt{real},\mathtt{real}]\rightarrow\mathtt{bool}]) \quad = \quad \mathtt{CONTEXT}$$

■

The following rule handles theory declarations.

Definition 5.3 (type rule for contexts with theory declarations)

$$\begin{array}{rcl}
\tau(\Theta)(\Gamma, m{:}\,\mathtt{THEORY} = \Delta) & = & \mathtt{CONTEXT}\ \textit{if}\ \Theta(m), \Gamma(m)\ \textit{are undefined} \\
 & & \Delta\ \textit{only has constant and type declarations,} \\
 & & \tau(\Theta;\Gamma)(\Delta) = \mathtt{CONTEXT}, \\
 & & \tau(\Theta)(\Gamma) = \mathtt{CONTEXT}
\end{array}$$

■

Example 5.4 (contexts with theory declarations)

$$\begin{array}{l}
\tau(\Omega)(\mathtt{reals}{:}\,\mathtt{THEORY} = (\mathtt{real}{:}\,\mathtt{TYPE}, \mathtt{0}{:}\,\mathtt{real}, \leq{:}\,[[\mathtt{real},\mathtt{real}]\rightarrow\mathtt{bool}])) \\
\quad = \quad \mathtt{CONTEXT}
\end{array}$$

■

Any reference to a type name or a constant s declared in a theory m outside of this theory must be prefixed by the theory name, as in $m.s$. Note that references to a type name or constant that is declared in the same theory should not be given a theory prefix. Before we can give the type rules, we must update the definition of the type expansion operation δ to prefix symbols with their theory names. Let $\Gamma(m)(s)$ abbreviate *definition*$(\Gamma(m))(s)$, which is the declaration of the symbol s in the definition of the theory m. Let $\eta(\Gamma, m)(a)$ be the result of prefixing every unprefixed type or constant symbol in a by m, where a is either an individual or type expression. We omit the definition of η since it is straightforward.

We modify the definition of δ in Definition 2.16 with the following clauses.

Definition 5.5 (expanded type for prefixed symbols)

$$\begin{array}{rcl} \delta(\Gamma)(m.s) & = & \delta(\Gamma)(\eta(\Gamma, m)(\mathit{definition}(\Gamma(m)(s)))),\ \mathit{if} \\ & & \mathit{definition}(\Gamma(m)(s))\ \mathit{is\ nonempty.} \\ \delta(\Gamma)(m.s) & = & m.s\ \mathit{if\ definition}(\Gamma(m)(s))\ \mathit{is\ empty.} \end{array}$$

■

Example 5.6 (expanded type for prefixed symbols) Let Ω'' be the context

$$\begin{array}{rl} \Omega, \texttt{reals}\!: \texttt{THEORY} = & (\texttt{real}\!: \texttt{TYPE}, \\ & \texttt{0}\!: \texttt{real}, \\ & \leq\!: [[\texttt{real}, \texttt{real}] \to \texttt{bool}], \\ & \texttt{nonneg\_real}\!: \texttt{TYPE} = \{x\!: \texttt{real} \mid\ \leq (0, x)\}, \\ & \texttt{1}\!: \texttt{nonneg\_real}) \end{array}$$

$$\delta(\Omega'')(\texttt{reals.nonneg\_real}) = \{x\!: \texttt{reals.real} \mid \texttt{reals.} \leq (\texttt{reals.0}, x)\}$$

■

The type rules for prefixed symbols are given below.

Definition 5.7 (type rules for prefixed symbols)

$$\begin{array}{rcl} \tau(\Gamma)(m.s) & = & \texttt{TYPE},\ \mathit{if\ kind}(\Gamma(m)) = \texttt{THEORY}\ \mathit{and} \\ & & \mathit{kind}(\Gamma(m)(s)) = \texttt{TYPE} \\ \tau(\Gamma)(m.s) & = & \delta(\Gamma)(\eta(\Gamma, m)(\mathit{type}(\Gamma(m)(s)))), \\ & & \mathit{if\ kind}(\Gamma(m)) = \texttt{THEORY}\ \mathit{and} \\ & & \mathit{kind}(\Gamma(m)(s)) = \texttt{CONSTANT} \end{array}$$

■

Example 5.8 (type rules for prefixed symbols)

$$\begin{aligned}\tau(\Omega'')(\texttt{reals.nonneg\_real}) &= \texttt{TYPE}\\ \tau(\Omega'')(\texttt{reals.1}) &= \{x{:}\,\texttt{reals.real} \mid \texttt{reals.}\leq(\texttt{reals.0}, x)\}\end{aligned}$$

■

The operations π, and μ remain unchanged. An assignment γ now maps a theory name m to an assignment $\gamma(m)$.

Definition 5.9 (meaning function for prefixed symbols)

$$\mathcal{M}(\Gamma \mid \gamma)(m.s) \;=\; \gamma(m)(s)$$

■

Example 5.10 (meaning function for prefixed symbols) Let ω'' be a satisfying assignment for Ω'' of the form

$$\ldots\{\texttt{reals} \leftarrow \{\texttt{real} \leftarrow \mathbf{R}\}\{\texttt{0} \leftarrow \mathbf{0}\}\ldots\}\ldots.$$

$$\begin{aligned}\mathcal{M}(\Omega'' \mid \omega'')(\texttt{reals.real}) &= \mathbf{R}\\ \mathcal{M}(\Omega'' \mid \omega'')(\texttt{reals.0}) &= \mathbf{0}\end{aligned}$$

■

Definition 5.11 (satisfaction for contexts with theories) *An assignment γ satisfies a context Γ if in addition to the constraints stated in Definition 2.18, γ maps every theory m declared in Γ to a satisfying assignment for the body of the theory given by definition$(\Gamma(m))$, that is for each declared symbol s in m:*

1. *If kind$(\Gamma(m)(s)) = \texttt{TYPE}$, then $\gamma(m)(s) \in U$.*
2. *If kind$(\Gamma(m)(s)) = \texttt{CONSTANT}$, then $\gamma(m)(s) \in \mathcal{M}(\Gamma \mid \gamma)(\tau(\Gamma)(m.s))$.*
3. *If definition$(\Gamma(m)(s))$ is nonempty, then*

$$\gamma(m)(s) = \mathcal{M}(\Gamma|\gamma)(\eta(\Gamma, m)(\mathit{definition}(\Gamma(m)(s)))).$$

■

5.2 Constant Definitions

We first extend the subset of PVS described so far to include constant definitions in a manner similar to type definitions. This extension is used in formalizing the semantics of parametric theories. The syntax for a constant definition is $c\colon T = a$ where *definition*$(\Gamma(c))$ is a. These definitions are explicit, that is, not recursive. With this extension, the type rule for constant declarations in contexts changes from that of Definition 3.10.

Definition 5.12 (type rule with constant definitions)

$$\begin{array}{lcl}\tau(\Theta)(\Gamma, c\colon T = a) & = & T,\ \textit{if}\ \Gamma(c)\ \textit{is undefined},\\ & & \Theta(c)\ \textit{is undefined},\\ & & \tau(\Theta)(\Gamma) = \texttt{CONTEXT},\\ & & \tau(\Theta;\Gamma)(a) = T',\\ & & (T \sim T')_\Gamma,\\ & & \vdash_\Gamma \pi(T)(a)\end{array}$$

■

The notion of satisfaction must be extended from that of Definition 5.11 to ensure that an assignment for a defined constant satisfies the definition.

Definition 5.13 (satisfaction with constant definitions) *An assignment γ satisfies a context Γ if in addition to the conditions in Definition 5.11, whenever kind$(\Gamma(s))$ =* `CONSTANT` *and definition$(\Gamma(s))$ is nonempty, then $\gamma(s) = \mathcal{M}(\Gamma \mid \gamma)(\textit{definition}(\Gamma(s)))$.* ■

5.3 Parametric Theories

The extension to parametric theories is obtained by permitting theories to be declared as $m[\Pi]\colon \texttt{THEORY} = \Delta$, where Π is a context listing the parameters and Δ is the body of the theory. If the above declaration of m occurs in context Γ, then Π is *formals*$(\Gamma(m))$, and Δ is *definition*$(\Gamma(m))$. For nonparametric theories, *formals*$(\Gamma(m))$ is empty. Types or constants declared in a parametric theory are referenced outside the theory as $m[\sigma].s$, where σ is a list of *actual* parameters consisting of types and terms. The type rule from the nonparametric case must be modified to check the parameters.

Definition 5.14 (type rule for contexts with parametric theories)

$$\begin{array}{ll}\tau(\Theta)(\Gamma, m[\Pi]\colon \texttt{THEORY} = \Delta) & \\ \quad = & \texttt{CONTEXT}\ \textit{if}\ \Gamma(m), \Theta(m), \Pi(m)\ \textit{are undefined} \\ & \tau(\Theta)(\Gamma) = \texttt{CONTEXT} \\ & \tau(\Theta;\Gamma)(\Pi) = \texttt{CONTEXT}, \\ & \Pi\ \textit{has only constant and} \\ & \textit{type declarations without definitions,} \\ & \tau(\Theta;\Gamma;\Pi)(\Delta) = \texttt{CONTEXT} \\ & \ \Delta\ \textit{only has type and constant declarations}\end{array}$$

■

The type rules for prefixed symbols are given below. The notation $\Pi = \sigma$, where Π is of the form $s_1\colon \alpha_1, \ldots, s_n\colon \alpha_n$, and σ is of the form $\sigma_1, \ldots, \sigma_n$, is short for the context $s_1\colon \alpha_1 = \sigma_1, \ldots, s_n\colon \alpha_n = \sigma_n$. The definition of η is now extended to substitute actual theory parameters for formals, so that $\eta(\Gamma, m[\sigma])(a)$ prefixes every unprefixed symbol s in a that is declared in *definition*$(\Gamma(m))$ by $m[\sigma]$, and replaces any s_i in a that is declared in *formals*$(\Gamma(m))$ by the corresponding σ_i in σ.

Definition 5.15 (type rules for prefixed names with actuals) *Let* Π *be formals*$(\Gamma(m))$.

$$\begin{array}{lcl}\tau(\Gamma)(m[\sigma].s) & = & \texttt{TYPE},\ \textit{if} \\ & & \textit{kind}(\Gamma(m)) = \texttt{THEORY} \\ & & \textit{kind}(\Gamma(m))(s) = \texttt{TYPE}\ \textit{and} \\ & & \tau(\Gamma)(\Pi = \sigma) = \texttt{CONTEXT} \\ \tau(\Gamma)(m[\sigma].s) & = & \delta(\Gamma)((\eta(\Gamma, m[\sigma])(\textit{type}(\Gamma(m)(s)))), \\ & & \ \textit{if kind}(\Gamma(m)) = \texttt{THEORY} \\ & & \textit{kind}(\Gamma(m)(s)) = \texttt{CONSTANT}\ \textit{and} \\ & & \tau(\Gamma)(\Pi = \sigma) = \texttt{CONTEXT}\end{array}$$

■

Definition 5.16 (type expansion with parametric theories)

$$\begin{array}{lcl}\delta(\Gamma)(m[\sigma].s) & = & \delta(\Gamma)((\eta(\Gamma, m[\sigma])(\textit{definition}(\Gamma(m)(s))))),\ \textit{if} \\ & & \textit{definition}(\Gamma(m)(s))\ \textit{is nonempty.} \\ \delta(\Gamma)(m[\sigma].s) & = & m[\sigma].s,\ \textit{if definition}(\Gamma(m)(s))\ \textit{is empty.}\end{array}$$

■

The definition of an assignment for a context with parametric theories is a bit complicated. In the nonparametric case, $\gamma(m)$ simply returns an assignment of values for the types and constants declared in the theory m. For the case of parametric theories m, $\gamma(m)$ returns a function that maps the meaning of the given actuals σ to an assignment $\gamma(m)(\mathcal{M}(\Gamma \mid \gamma)(\sigma))$ for the types and constants declared in the theory m. There is an important restriction that $\gamma(m)$ must be *rank-preserving*, that is, if ϖ and ϖ' are assignments for Π so that for each i where Π_i is a type parameter, the rank of $\varpi(\Pi_i)$ equals the rank of $\varpi'(\Pi_i)$, then the ranks of $\gamma(m)(\varpi)(s)$ and $\gamma(m)(\varpi')(s)$ must be the same for each type symbol s declared in m.

It is also important to observe that the semantics of parametric theories makes use of the axiom of choice since the assignment corresponding to a theory `m` of the form `m[t: TYPE]: THEORY` $=$ `{c: t}` is essentially a choice function.

Let $\gamma\{\Pi \leftarrow \varpi\}$ represent the assignment such that $\gamma\{\Pi \leftarrow \varpi\}(s) = \varpi(s)$ for s in the domain of the context Π, and $\gamma(s)$, otherwise. The meaning of symbols of the form $m[\sigma].s$ can then be defined as below.

Definition 5.17 (meaning function for prefixed symbols with actuals)

$$\begin{aligned}
&\mathcal{M}(\Gamma \mid \gamma)(m[\sigma].s)\\
&\quad = \mathcal{M}(\Gamma; \Pi; \Delta \mid \gamma\{\Pi \leftarrow \varpi\}\{\Delta \leftarrow \gamma(m)(\varpi)\})(s), \ \textit{where}\\
&\qquad \Pi = \textit{formals}(\Gamma(m))\\
&\qquad \Delta = \textit{definition}(\Gamma(m))\\
&\qquad \varpi(r) = \mathcal{M}(\Gamma \mid \gamma)((\Pi = \sigma)(r)), \ \textit{for } r \in \Pi
\end{aligned}$$

■

The definition of a satisfying assignment given in Definition 5.11 also must be strengthened. Let Π be the formal parameters to theory m in context Γ; then, an assignment ϖ is said to be satisfying *parameter assignment* for Π under the assignment γ to Γ iff $\gamma\{\Pi \leftarrow \varpi\}$ is a satisfying assignment for Π.

Definition 5.18 (satisfaction for contexts with parametric theories) *An assignment γ satisfies a context Γ if in addition to the constraints stated in Definition 5.11, γ maps every parametric theory m declared in Γ with parameters Π and definition Δ, to a function that maps any satisfying parameter assignment ϖ for the theory parameters Π (namely, formals$(\Gamma(m))$) to a satisfying assignment $\gamma\{\Pi \leftarrow \varpi\}\{\Delta \leftarrow \gamma(m)(\varpi)\}$ for Δ (given by definition$(\Gamma(m))$).* ■

5.4 Summary

Theories are used to package related declarations together. Parametric theories can be used to package together declarations that are generic in type and individual parameters. The type rules for contexts must be extended to accommodate the theories. The type rules for simple (nonparametric) theories are straightforward given this extension. The operation of expanding a type using type definitions must be enhanced so that symbols declared in a theory are prefixed with their theory name when referenced outside the theory. Assignments now have the same nested structure as contexts, and the semantic definition is easily extended to handle prefixed symbols. Parametric theories are more complex. The theory prefixes now contain actual parameters that have to be typechecked relative to the expected formal parameters. The assignments corresponding to parametric theories are functions that map given assignments for the formals to assignments for the declarations within a theory. Such a mapping must be constrained to be rank-preserving. Parametric theories can have subtype parameters, and assumptions on the parameters. The rules for subtype parameters and assumptions are omitted for now but will be included in an expanded version of this report.

Chapter 6

Conditional Expressions and Logical Connectives

We have, so far, introduced the core of PVS containing types, type definitions, constant and variable declarations, subtypes, dependent types, and theories. In extending the language with both explicit and recursive constant definitions and formulas, a crucial difference is that the logical context under which a type-correctness condition is generated provides additional assumptions that can be used in proving any proof obligations. Examples of expressions where an extended context is needed to establish type correctness by discharging proof obligations include

1. $x \neq y \supset (x+y)/(x-y) \leq 0$. The type of the division operator constrains the denominator to be nonzero, that is, $\{x\colon \texttt{real} \mid x \neq 0\}$. In the given expression, the denominator can be shown to be nonzero only in the context of the antecedent $x \neq y$.

2. $\texttt{IF}(i > 0, i, -i)$ has type `nat` given integer i provided the *then* and *else* parts are typechecked with the assumptions $i > 0$ and $\neg(i > 0)$, respectively.

PVS has a polymorphic primitive equality predicate:

```
equality[T : TYPE] : THEORY = { =: [[T, T] -> bool] }
```

Note that an equality of the form $\texttt{equality}[T].{=}(a, b)$ is informally written as $a = b$. When it is relevant to indicate the type parameter, we write the equality as $a =_T b$. It can be deduced from the meaning of equality that if S is a subtype of T, then for a and b in S, it must be the case that $a =_S b$

iff $a =_T b$. Thus, we can assume that equality is always parameterized by a maximal type. We assume that any relevant context Γ contains the above declaration of the theory `equality`. Furthermore, any satisfying assignment γ for such a Γ must satisfy

$$\gamma(\texttt{equality})(X)(=) \quad = \quad \{\langle x, x\rangle \mid x \in X\}.$$

The negation operation can be defined in terms of equality as shown below. We assume that the context contains a declaration of the form

$$\neg \ : \ [\texttt{bool}{\rightarrow}\texttt{bool}] \quad = \quad (\lambda \ (x \ : \ \texttt{bool}){:} \ x \ = \ \texttt{FALSE})$$

As is clear, a satisfying assignment γ for a context Γ containing the above declaration must be such that $\gamma(\neg)$ yields the usual truth-table semantics, that is, $\{\langle \mathbf{0}, \mathbf{1}\rangle, \langle \mathbf{1}, \mathbf{0}\rangle\}$.

We can then introduce the polymorphic `IF-THEN-ELSE` operation as follows:

```
if_def [T: TYPE]: THEORY = { IF:[bool,T,T -> T] }
```

In typechecking conditional expressions, the notion of context has to be extended to include formulas so that the typechecking of the subterm b in $\texttt{IF}(a, b, c)$ is done in the context of a, and the typechecking of c is done in the context of $\neg a$. There is one new typechecking rule for contexts with formulas.

$$\begin{aligned} \tau()(\Gamma, a) \quad = \quad & \texttt{CONTEXT}, \text{ if} \\ & \tau()(\Gamma) = \texttt{CONTEXT}, \text{ and} \\ & (\tau(\Gamma)(a) \sim \texttt{bool})_\Gamma \end{aligned}$$

Note that the type rule checks that the type of a is compatible with `bool` rather than equivalent to it since it is possible that the type of a might be a subtype of `bool`.

Definition 6.1 (satisfaction for contexts with formulas) *An assignment γ satisfies context Γ when in addition to the conditions in Definition 5.18, for each prefix Γ', a of Γ, $\mathcal{M}(\Gamma' \mid \gamma)(a) = \mathbf{1}$.* ■

The typechecking of conditional expressions is different from that of other application expressions since the *test* part of the conditional expression is introduced into the context as a contextual assumption.

Definition 6.2 (type rule for conditional expressions)

$$\begin{array}{rcl}\tau(\Gamma)(\texttt{if\_def}[T].\texttt{IF}(a,b,c)) & = & T,\ \textit{if}\ (\tau(\Gamma)(a) \sim \texttt{bool})_{\Gamma},\\ & & \tau(\Gamma,a)(b) = B,\\ & & (B \sim T)_{\Gamma,a},\\ & & \vdash_{\Gamma,a} \pi(T)(b)\\ & & \tau(\Gamma,\neg a)(c) = C,\\ & & (C \sim T)_{\Gamma,\neg a},\\ & & \vdash_{\Gamma,\neg a} \pi(T)(c)\end{array}$$

■

The meaning of conditional expressions must be treated in a special way since the *else* part need not denote when the *test* part is true and, correspondingly, the *then* part need not denote if the *test* part is false. We assume that any relevant contexts Γ contain the above declaration of the `if_def` theory. Conditional expressions can be regarded as a new construct in the language rather than a form of application. However, it is conservative to regard conditional expressions as applications since the latter introduce the additional constraint that all the arguments must already denote, that is, applications are *strict*.

Definition 6.3 (meaning function for conditional expressions)

$$\mathcal{M}(\Gamma \mid \gamma)(\texttt{if\_def}[T].\texttt{IF}(a,b,c)) \quad = \quad \begin{cases} \mathcal{M}(\Gamma \mid \gamma)(b), & \textit{if}\ \mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}\\ \mathcal{M}(\Gamma \mid \gamma)(c), & \textit{otherwise}\end{cases}$$

■

The semantics for conditional expressions raises an important issue. The equality

$$\texttt{if\_def}[\texttt{bool}].\texttt{IF}(x, y, \texttt{FALSE}) = \texttt{if\_def}[\texttt{bool}].\texttt{IF}(y, x, \texttt{FALSE})$$

is semantically valid for variables x and y of type `bool`. An expression like $\texttt{if\_def}[\texttt{bool}].\texttt{IF}(i \neq 0, 1/i > 0, \texttt{FALSE})$ can be typechecked to have the type `bool` since it generates a valid proof obligation $i \neq 0 \supset i \neq 0$, but the seemingly equivalent expression $\texttt{if\_def}[\texttt{bool}].\texttt{IF}(1/i > 0, i \neq 0, \texttt{FALSE})$ generates an unverifiable proof obligation $i \neq 0$. This may seem contradictory since the equality suggests a transformation of a type correct conditional expression to a type incorrect expression. The resolution here is that equality cannot be instantiated with $i \neq 0$ for x and $1/i > 0$ for y since the expression $1/i > 0$ typechecks as having type `bool` only when $i \neq 0$ is known from the context. The same applies in the case of the other propositional connectives, thus ensuring that each expression is type correct in the context in which it occurs.

We can then define the propositional connectives in terms of conditional expressions.

$$\begin{array}{lcl}
\wedge\colon [[\texttt{bool}, \texttt{bool}]\to\texttt{bool}] & = & \lambda(x\colon \texttt{bool}, y\colon \texttt{bool})\colon \texttt{if\_def}[\texttt{bool}].\texttt{IF}(x, y, \texttt{FALSE}) \\
\vee\colon [[\texttt{bool}, \texttt{bool}]\to\texttt{bool}] & = & \lambda(x\colon \texttt{bool}, y\colon \texttt{bool})\colon \texttt{if\_def}[\texttt{bool}].\texttt{IF}(x, \texttt{TRUE}, y) \\
\supset\colon [[\texttt{bool}, \texttt{bool}]\to\texttt{bool}] & = & \lambda(x\colon \texttt{bool}, y\colon \texttt{bool})\colon \texttt{if\_def}[\texttt{bool}].\texttt{IF}(x, y, \texttt{TRUE})
\end{array}$$

In the typechecking of terms of the form $a \wedge b$, we follow the corresponding rule for the definition so that the term a is assumed in the context when typechecking term b. Similarly, for $a \vee b$, the formula $\neg a$ is assumed in the context when typechecking b, and for $a \supset b$, the formula a is assumed in the context when typechecking b. The Boolean equivalence operator `IFF` has no special rules for adding formulas to contexts during typechecking.

6.1 Summary

The use of assumption formulas enables expressions to be typechecked within the narrow context of their use so that the governing assumptions can be used in discharging any proof obligations. The type rules for conditional expressions and the Boolean connectives $\wedge$, $\vee$, and $\supset$ make use of contextual assumptions.

Chapter 7

Proof Theory of PVS

The final step in the presentation of the semantics is the presentation of the proof rules for the idealized subset of PVS described thus far. As already indicated, the proof theory is an integral part of the semantics since typechecking and proof checking are closely intertwined. Fortunately, the proof rules turn out to be much less complicated than the type rules.

The PVS proof theory is presented in terms of a sequent calculus. A sequent is of the form $\Sigma \vdash_\Gamma \Lambda$, where Γ is the context, Σ is a *set* of *antecedent* formulas, and Λ is a *set* of *consequent* formulas. Such a sequent should be read as stating that the conjunction of the formulas in Σ implies the disjunction of formulas in Λ.

Inference rules are presented in the form

$$\frac{\textit{premise(s)}}{\textit{conclusion}}\ \textit{name} \qquad \textit{side condition}$$

7.1 PVS Proof Rules

7.1.1 Structural Rules

The structural rules permit the sequent to be rearranged or weakened via the introduction of new sequent formulas into the conclusion. All the structural rules can be expressed in terms of the single powerful weakening rule shown below. It allows a weaker statement to be derived from a stronger one by adding either antecedent formulas or consequent formulas. The relation $\Sigma_1 \subseteq \Sigma_2$ holds between two lists when all the formulas in Σ_1 occur in the list Σ_2.

$$\frac{\Sigma_1 \vdash_\Gamma \Lambda_1}{\Sigma_2 \vdash_\Gamma \Lambda_2}\ \mathbf{W} \qquad \text{if } \Sigma_1 \subseteq \Sigma_2 \text{ and } \Lambda_1 \subseteq \Lambda_2$$

Both the Contraction and Exchange rules shown below are absorbed by the above weakening rule **W**. The *Contraction* rules $\mathbf{C}\vdash$ and $\vdash\mathbf{C}$ allow multiple occurrences of the same sequent formula to be replaced by a single occurrence.

$$\frac{a, a, \Sigma \vdash_\Gamma \Lambda}{a, \Sigma \vdash_\Gamma \Lambda}\ \mathbf{C}\vdash \qquad\qquad \frac{\Sigma \vdash_\Gamma a, a, \Lambda}{\Sigma \vdash_\Gamma a, \Lambda}\ \vdash\mathbf{C}$$

The *Exchange* rule asserts that the order of the formulas in the antecedent and the consequent parts of the sequent is immaterial. It can be stated as

$$\frac{\Sigma_1, b, a, \Sigma_2 \vdash_\Gamma \Lambda}{\Sigma_1, a, b, \Sigma_2 \vdash_\Gamma \Lambda}\ \mathbf{X}\vdash \qquad\qquad \frac{\Sigma \vdash_\Gamma \Lambda_1, b, a, \Lambda_2}{\Sigma \vdash_\Gamma \Lambda_1, a, b, \Lambda_2}\ \vdash\mathbf{X}$$

As seen above, *inference rules* have the general form

$$\frac{\Sigma_1 \vdash \Lambda_1 \quad \cdots \quad \Sigma_n \vdash \Lambda_n}{\Sigma \vdash \Lambda}\ \mathbf{R}$$

This says that if we are given a leaf of a proof tree of the form $\Sigma \vdash \Lambda$, then by applying the rule named **R**, we may obtain a tree with n new leaves.

7.1.2 Cut Rule

The cut rule **Cut** can be used to introduce a case split on a formula a into a proof of a sequent $\Sigma \vdash_\Gamma \Lambda$ so as to yield the subgoals $\Sigma, a \vdash_\Gamma \Lambda$ and $\Sigma \vdash_\Gamma a, \Lambda$, which can be seen as assuming a along one branch and $\neg a$ along the other.

$$\frac{(\tau(\Gamma)(a) \sim \texttt{bool})_\Gamma \qquad \Sigma, a \vdash_\Gamma \Lambda \qquad \Sigma \vdash_\Gamma a, \Lambda}{\Sigma \vdash_\Gamma \Lambda}\ \mathbf{Cut}$$

7.1.3 Propositional Axioms

The axioms rule **Ax** simply asserts that a follows from a.

$$\frac{}{\Sigma, a \vdash_\Gamma a, \Lambda}\ \mathbf{Ax}$$

The next two rules assert that any sequent with either an antecedent occurrence of `FALSE` or a consequent occurrence of `TRUE` is an axiom.

$$\frac{}{\Sigma, \texttt{FALSE} \vdash_\Gamma \Lambda}\ \texttt{FALSE}\vdash \qquad\qquad \frac{}{\Sigma \vdash_\Gamma \texttt{TRUE}, \Lambda}\ \vdash\texttt{TRUE}$$

7.1.4 Context Rules

Certain formulas hold in a context simply because they are already asserted in the context either as a formula or a constant definition.

$$\frac{}{\vdash_{\Gamma} a}\ \textbf{ContextFormula} \quad \text{if } a \text{ is a formula in } \Gamma$$

$$\frac{}{\vdash_{\Gamma} s = a}\ \textbf{ContextDefinition} \text{ if } s\colon T = a \text{ is a constant definition in } \Gamma$$

The context Γ can be extended with antecedent formulas or negations of consequent formulas using the following two rules.

$$\frac{\Sigma, a \vdash_{\Gamma, a} \Lambda}{\Sigma, a \vdash_{\Gamma} \Lambda}\ \textbf{Context} \vdash \qquad \frac{\Sigma \vdash_{\Gamma, \neg a} a, \Lambda}{\Sigma \vdash_{\Gamma} a, \Lambda} \vdash \textbf{Context}$$

The following context-weakening rule is useful since it shows that provability is monotonic with respect to the context.

$$\frac{\Sigma \vdash_{\Gamma} \Lambda}{\Sigma \vdash_{\Gamma'} \Lambda}\ \textbf{ContextW} \quad \text{if } \Gamma \text{ is a prefix of } \Gamma'$$

7.1.5 Conditional Rules

The rules governing the elimination of `IF-THEN-ELSE` in a proof are unusual since they augment the context with the test part or its negation, as in the corresponding type rules.

$$\frac{\Sigma, a, b \vdash_{\Gamma, a} \Lambda \qquad \Sigma, c \vdash_{\Gamma, \neg a} a, \Delta}{\Sigma, \texttt{IF}(a, b, c) \vdash_{\Gamma} \Lambda}\ \texttt{IF} \vdash$$

$$\frac{\Sigma, a \vdash_{\Gamma, a} b, \Lambda \qquad \Sigma \vdash_{\Gamma, \neg a} a, c, \Lambda}{\Sigma \vdash_{\Gamma} \texttt{IF}(a, b, c), \Lambda} \vdash \texttt{IF}$$

7.1.6 Equality Rules

The rules for equality can be stated as below. The rules of transitivity and symmetry for equality can be derived from these rules. The notation $a[e]$ is used to highlight one or more occurrences of e in the formula a such that there are no free variable occurrences in e.[1] The notation $\Lambda[e]$ similarly highlights occurrences of e in Λ.

[1] We enforce an invariant on a sequent that it must not contain any free variables. This invariant is preserved by each of the proof rules.

$$\frac{}{\Sigma \vdash_\Gamma a = a, \Lambda}\ \textbf{Refl} \qquad \frac{a = b, \Sigma[b] \vdash_\Gamma \Lambda[b]}{a = b, \Sigma[a] \vdash_\Gamma \Lambda[a]}\ \textbf{Repl}$$

7.1.7 Boolean Equality Rules

The rule **Repl** `TRUE` asserts that an antecedent formula a can be treated as an antecedent equality of the form $a = \texttt{TRUE}$, and correspondingly, a consequent formula a can be treated as an antecedent equality of the form $a = \texttt{FALSE}$.

$$\frac{\Sigma[\texttt{TRUE}], a \vdash_\Gamma \Lambda[\texttt{TRUE}]}{\Sigma[a], a \vdash_\Gamma \Lambda[a]}\ \textbf{Repl}\ \texttt{TRUE} \qquad \frac{\Sigma[\texttt{FALSE}], a \vdash_\Gamma \Lambda[\texttt{FALSE}]}{\Sigma[a] \vdash_\Gamma a, \Lambda[a]}\ \textbf{Repl}\ \texttt{FALSE}$$

The rule `TRUE-FALSE` asserts that `TRUE` and `FALSE` are distinct Boolean constants.

$$\frac{}{\Sigma, \texttt{TRUE} = \texttt{FALSE} \vdash_\Gamma \Lambda}\ \texttt{TRUE-FALSE}$$

7.1.8 Reduction Rules

The reduction rules are equality rules (axioms) that provide the obvious simplifications for applications involving lambda abstractions and product projections.

$$\frac{}{\vdash_\Gamma (\lambda(x\colon T)\colon\ a)(b) = a[b/x]}\ \beta$$

$$\frac{}{\vdash_\Gamma \mathtt{p}_i(a_1, a_2) = a_i}\ \pi$$

7.1.9 Extensionality Rules

The extensionality rules are also equality rules for establishing equality between two expressions of function or product type. The extensionality rule for functions, **FunExt**, introduces a Skolem constant s to determine that two functions f and g are equal when the results of applying them to an arbitrary argument s are equal.

$$\frac{\Sigma \vdash_{\Gamma, s:A} (f\ s) =_{B[s/x]} (g\ s), \Lambda}{\Sigma \vdash_\Gamma f =_{[x:A \to B]} g, \Lambda}\ \textbf{FunExt} \qquad \Gamma(s)\ \text{undefined}$$

The extensionality rule for products asserts that two products are equal if their corresponding projections are equal.

$$\frac{\Sigma \vdash_\Gamma \mathtt{p}_1(a) =_{T_1} \mathtt{p}_1(b), \Lambda \qquad \Sigma \vdash_\Gamma \mathtt{p}_2(a) =_{T_2[(\mathtt{p}_1\ a)/x]} \mathtt{p}_2(b), \Lambda}{\Sigma \vdash_\Gamma a =_{[x:T_1 T_2]} b, \Lambda}\ \textbf{TupExt}$$

Recall that the quantifiers can be defined in terms of lambda abstraction and equality so that $(\forall(x{:}\,T){:}\,a)$ is just $(\lambda(x{:}\,T){:}\,a) = (\lambda(x{:}\,T){:}\,\mathtt{TRUE})$. Existential quantification $(\exists(x{:}\,T){:}\,a)$ can easily be defined as $\neg(\forall(x{:}\,T){:}\,\neg a)$. The proof rules for quantifiers can then be derived from the rules β, **TupExt**, and the equality rules.

7.1.10 Type Constraint Rule

We need a rule to introduce the type constraint on a term as an antecedent formula of the given goal sequent.

$$\frac{\tau(\Gamma)(a) = A \qquad \pi(A)(a), \Sigma \vdash_\Gamma \Lambda}{\Sigma \vdash_\Gamma \Lambda}\ \textbf{Typepred}$$

7.2 Soundness of the Proof Rules

Proposition 7.1 *If Γ is a prefix of Γ', $\tau()(\Gamma) = \tau()(\Gamma') = \mathtt{CONTEXT}$, γ' is a satisfying assignment for Γ', and $\gamma = \gamma' \upharpoonright \Gamma$ then for any a such that $\tau(\Gamma)(a) = \tau(\Gamma')(a)$, it is the case that $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathcal{M}(\Gamma' \mid \gamma')(a)$.*

Theorem 7.2 (soundness) *If $\tau()(\Gamma) = \mathtt{CONTEXT}$ such that for every formula a in $\Sigma; \Lambda$, $(\tau(\Gamma)(a) \sim \mathtt{bool})_\Gamma$, and $\Sigma \vdash_\Gamma \Lambda$ is provable, then for any satisfying assignment γ for Γ, either there is a formula b in Σ, such that $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{0}$ or a formula c in Λ, such that $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$.*

Proof. The proof is by induction on the structure of the proof of $\Sigma \vdash_\Gamma \Lambda$. Recall that this proof is actually part of a simultaneous induction that includes the soundness of the type rules relative to the semantic function, that is, Theorems 4.14 and 4.18. Specific invocations of the soundness theorem occur in the proofs of Theorem 3.19 and Proposition 4.17.

1. *Structural Rules*: Since the subset of formulas in the premise and the conclusion of these rules are the same, the conclusion follows easily from the induction hypothesis.

2. *Cut*: By the semantic soundness of the type rules, we have $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathbf{2}$. If $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{0}$, then by the induction hypothesis on the second subgoal of the proof rule, there must be some b in Σ such that $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{0}$ or a c in Λ such that $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$. The case when $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}$ is symmetrical.

3. *Propositional Axioms*: Obvious.

4. *Context Rules*:

   ContextFormula: If γ satisfies Γ and $a \in \Gamma$, then $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}$.

   ContextDefinition: If γ satisfies Γ and $s\!:\!T = a$ is a declaration in Γ, then by the definition of satisfaction, $\mathcal{M}(\Gamma \mid \gamma)(s) = \mathcal{M}(\Gamma \mid \gamma)(a)$.

   Context $\vdash$: The argument is trivial when $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{0}$. Otherwise, γ satisfies the extended context Γ, a, and the conclusion follows from the induction hypothesis.

   $\vdash$ Context: Similar to **Context** $\vdash$ above.

   ContextW: If γ satisfies Γ', then it also satisfies Γ, and hence the proof.

5. *Conditional Rules*: We only consider `IF` $\vdash$ since the $\vdash$ `IF` proof is similar. If $\mathcal{M}(\Gamma \mid \gamma)(\texttt{IF}(a, b, c)) = \mathbf{0}$, the conclusion follows trivially. Otherwise, If γ satisfies Γ, then $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathbf{2}$. If $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}$, then $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{1}$. The induction hypothesis on the subgoal $\Sigma, a, b \vdash_{\Gamma, a} \Lambda$ yields the desired conclusion. Similarly, if $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{0}$, we have $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$ and the induction hypothesis on the second subgoal yields the desired conclusion.

6. *Equality Rules*: The **Refl** rule is obvious. For the **Repl** rule, if $\mathcal{M}(\Gamma \mid \gamma)(a = b) = \mathbf{0}$, the conclusion follows trivially. Otherwise, $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathcal{M}(\Gamma \mid \gamma)(b)$. Hence, γ satisfies the extended context $\Gamma, a = b$. Then for each $c[a]$ in $\Sigma[a]$ or $\Lambda[a]$, $\mathcal{M}(\Gamma \mid \gamma)(c[a]) = \mathcal{M}(\Gamma \mid \gamma)(c[b])$.

7. *Boolean Equality Rules*: The **Repl** `TRUE` and **Repl** `FALSE` rules follow easily since when $\mathcal{M}(\Gamma \mid \gamma)(a) = \mathbf{1}$, we have $\mathcal{M}(\Gamma \mid \gamma)(c[a]) = \mathcal{M}(\Gamma \mid \gamma)(c[\texttt{TRUE}])$. A similar argument applies to **Repl** `FALSE`.

   The soundness of `TRUE-FALSE` is easy since $\mathcal{M}(\Gamma \mid \gamma)(\texttt{TRUE} = \texttt{FALSE}) = \mathbf{0}$.

8. *Reduction Rules*: The β-reduction rule follows because $\mathcal{M}(\Gamma \mid \gamma)((\lambda(x{:}\,T){:}\ a)(b))$ is $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ T \mid \gamma\{x \leftarrow \mathcal{M}(\Gamma \mid \gamma)(b)\})(a)$ which by the Substitution Lemma 4.16 is equal to $\mathcal{M}(\Gamma \mid \gamma)(a[b/x])$.

   The soundness π-reduction rule is a direct consequent of Definition 2.6.

9. *Extensionality Rules*:

   FunExt: First consider the case when the domain type $\mathcal{M}(\Gamma \mid \gamma)(A)$ is empty. Then by Definition 4.12, $\mathcal{M}(\Gamma \mid \gamma)(f) = \mathcal{M}(\Gamma \mid \gamma)(g) = \emptyset$. Therefore $\mathcal{M}(\Gamma \mid \gamma)(f = g) = \mathbf{1}$ and hence the conclusion.[2]

   The case when $\mathcal{M}(\Gamma \mid \gamma)(A)$ is nonempty, we have for any γ satisfying Γ and $s \in \mathcal{M}(\Gamma \mid \gamma)(A)$, that γ' given by $\gamma\{s \leftarrow z\}$ is a satisfying assignment for $\Gamma, s{:}\,A$. By the induction hypothesis, there is either an a in Σ such that $\mathcal{M}(\Gamma, s{:}\,A \mid \gamma')(b) = \mathbf{0}$ or a c in $(f\ s) = (g\ s), \Lambda$ such that $\mathcal{M}(\Gamma, s{:}\,A \mid \gamma')(c) = \mathbf{1}$. If we have such a b in Σ, by Proposition 7.1, we also have that $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{0}$. A similar argument can be used if we have such a c in Λ. If c is $(f\ s) = (g\ s)$, then $\mathcal{M}(\Gamma \mid \gamma)(f)(z) = \mathcal{M}(\Gamma \mid \gamma)(g)(z)$ for every z in $\mathcal{M}(\Gamma \mid \gamma)(A)$. By set-theoretic extensionality, this means that $\mathcal{M}(\Gamma \mid \gamma)(f)$ and $\mathcal{M}(\Gamma \mid \gamma)(g)$ are identical elements of ΠF where F maps z in $\mathcal{M}(\Gamma \mid \gamma)(A)$ to an element of $\mathcal{M}(\Gamma, x{:}\,\texttt{VAR}\ A \mid \gamma\{x \leftarrow z\})(B)$. Therefore $\mathcal{M}(\Gamma \mid \gamma)(f = g) = \mathbf{1}$ as desired.

   TupExt: If there is some d in Σ such that by applying the induction hypothesis to any of the subgoals $\mathcal{M}(\Gamma \mid \gamma)(d) = \mathbf{0}$, then the same holds for the conclusion sequent. Similarly, if the induction hypothesis on some subgoal yields a c in Λ such that $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$, then the same holds for the conclusion sequent. So the remaining case is when, by the induction hypothesis, $\mathcal{M}(\Gamma \mid \gamma)(\mathtt{p}_i(a)) = \mathcal{M}(\Gamma \mid \gamma)(\mathtt{p}_i(b))$ for each $i \in \{1, 2\}$. It is therefore easy to conclude by set-theoretic extensionality that $\mathcal{M}(\Gamma \mid \gamma)(a)$ and $\mathcal{M}(\Gamma \mid \gamma)(b)$ are identical elements of $\mathcal{M}(\Gamma \mid \gamma)(a/[T_1, T_2])$. We can then use Proposition 4.17 to conclude that $\mathcal{M}(\Gamma \mid \gamma)(a)$ and $\mathcal{M}(\Gamma \mid \gamma)(b)$ are identical elements of $\mathcal{M}(\Gamma \mid \gamma)([T_1, T_2])$.

10. *Type Constraint Rule*: Recall from Proposition 3.17 that when $\tau(\Gamma)(a) = A$, then $\mathcal{M}(\Gamma \mid \gamma)(\pi(A)(a)) = \mathbf{1}$. Given this and the induction hy-

[2]Since the subgoal sequent $\Sigma \vdash_{\Gamma, s:A} (f\ s) = (g\ s), \Lambda$ is valid when $\mathcal{M}(\Gamma \mid \gamma)(A) = \emptyset$ for all assignments γ, it is natural to ask how it is actually proved. The only way a type A can be empty under any assignment γ is if $\mathcal{M}(\Gamma \mid \gamma)(\pi(A)(a) = \mathbf{0})$. The **Typepred** rule can therefore be used on the Skolem constant s to complete the proof.

pothesis, it must either be the case that we have a b in Σ such that $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{0}$ or a c in Λ such that $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$.

■

To tie the development so far into a single simultaneous induction as promised, we state the key theorem whose subproofs have been given by the theorems presented thus far, namely, Theorems 4.14, 4.18, and 7.2.

Theorem 7.3 *If* $\tau()(\Gamma) = \texttt{CONTEXT}$*, then*

1. *If* Σ, Λ *is a list of preterms such that for every* a *in* $\Sigma; \Lambda$, $(\tau(\Gamma)(a) \sim \texttt{bool})_\Gamma$, *and* $\Sigma \vdash_\Gamma \Lambda$ *is provable, then for any satisfying assignment* γ *for* Γ, *either there is a* b *in* Σ, *such that* $\mathcal{M}(\Gamma \mid \gamma)(b) = \mathbf{0}$ *or a* c *in* Λ, *such that* $\mathcal{M}(\Gamma \mid \gamma)(c) = \mathbf{1}$.

2. *If* A *is a pretype such that* $\tau(\Gamma)(A) = \texttt{TYPE}$, *then for any assignment* γ *satisfying* Γ, $\mathcal{M}(\Gamma \mid \gamma)(A) \in U$.

3. *If* a *is a preterm such that* $\tau(\Gamma)(a) = A$, *then for any assignment* γ *satisfying* Γ, $\mathcal{M}(\Gamma \mid \gamma)(a) \in \mathcal{M}(\Gamma \mid \gamma)(A)$.

7.3 Summary

The logical inference rules for the PVS logic have been presented in a sequent calculus format. The formal semantics presented in the earlier chapters is used to establish the soundness of these proof rules.

Chapter 8

Conclusion

We have presented the syntax and semantics of idealized PVS in several stages. In the first stage we introduced the simply typed fragment, which was then extended with type definitions. The third such fragment included subtyping; the fourth fragment introduced dependent typing. Finally, we introduced constant definitions and parametric and nonparametric theories.

The semantic definition was given in a novel, functional style where a canonical type was assigned to each type correct term. The interplay between types and proofs in PVS introduced subtleties and complexities into the semantic definition. We can now answer some of the questions raised in Chapter 1:

- *What is the semantic core of the language, and what is just syntactic sugar?*

  The semantic core of the language is a typed lambda calculus with simple function and tuple types, predicate subtypes, dependent types, parametric theories, and conditional expressions. Many of the other features of the PVS language such as records and update expressions can be explained in terms of the core language.

- *What are the rules for determining whether a given PVS expression is well typed?*

  The typechecking rules have been presented in terms of the definition of the τ operator in Chapters 2, 3, 4, 5, and 6.

- *How is subtyping handled, and in particular, how are proof obligations corresponding to subtypes generated?*

  Typechecking an expression a with respect to predicate subtype constraint $\{x\colon T | p(x)\}$ is done by generating the proof obligation $p(a)$ under the logical context in which a is being typechecked. This is made precise in Definitions 3.10 and 6.2. Proof obligations are generated when typechecking contexts (for nonemptiness), typechecking expressions with respect to expected subtypes, and comparing two types containing subtype expressions for compatibility.

- *What is the meaning, in set-theoretic terms, of a PVS expression or assertion?*

  The set-theoretic meaning of well-formed PVS types and expressions is given by a meaning function $\mathcal{M}$ that assigns a set $\mathcal{M}(\Gamma \mid \gamma)(T)$ from the universe U to each type T, and an element $\mathcal{M}(\Gamma \mid \gamma)(a)$ of $\mathcal{M}(\Gamma \mid \gamma)(T)$ to a given term a of type T.

- *Are the type rules sound with respect to the semantics?*

  The typechecking function τ is defined to check contexts, preterms, and pretypes for type correctness. The type rules are shown to be sound with respect to the given semantics in Theorem 7.3.

- *Are the proof rules sound with respect to the semantics?*

  The proof rules are given in Chapter 7 in a sequent calculus format and proved to be sound with respect to the semantics in Theorem 7.3.

- *What is the form of dependent typing used by PVS, and what kinds of type dependencies are disallowed by the language?*

  The semantic analysis of dependent typing in Chapter 4 revealed that type dependencies were constrained to be rank-bounded. This is true because the dependencies in dependent typing only constrain the predicate part of predicate subtypes. Thus, when there is a dependent type $T(n)$ that depends on a parameter n, the meaning of $T(n)$ has a fixed rank regardless of the meaning assigned to n. The PVS language features used to define dependent types all preserve the rank-boundedness. Language extensions violating rank-boundedness such as a type dependency of the form $[n\colon nat \rightarrow T^n]$ are disallowed. One can extend the language with such dependent types, but the semantics would then be considerably more complicated.

- *What is the meaning of theory-level parametricity, and what, if any, are the semantic limits on such parameterization?*

The semantics of parametric theories is described in Chapter 5. In particular, the semantics for parametric theories is given in terms of rank-preserving maps between the meanings of the parameters and the meanings of the identifiers declared in the theory. These maps must be such that the rank of an assignment to a type in a theory depends only on the ranks of the (meanings of the) type parameters.

- *What language extensions are incompatible with the reference semantics given here?*

  We have already indicated that any language extension, such as an n-tuple type T^n, that violates rank-boundedness would be incompatible with the semantics presented here.

This report presents only the core language of PVS. A more complete semantic treatment would include arithmetic, recursive constant definitions, inductive definitions, recursive datatypes, assumptions on theory parameters, and type judgements.

Acknowledgments. The advice and encouragement of John Rushby, Rick Butler, Paul Miner, Pat Lincoln, and Mandayam Srivas are greatly appreciated, as are the useful expert comments of Peter Dybjer, Mike Gordon, Doug Howe, and Paul Jackson. Bruno Dutertre, Paul Miner, and Harald Rueß suggested numerous improvements to earlier drafts.

Bibliography

[AMCP84] P. B. Andrews, D. A. Miller, E. L. Cohen, and F. Pfenning. Automating higher-order logic. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, pages 169–192. American Mathematical Society, Providence, R.I., 1984.

[And86] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic Press, New York, NY, 1986.

[CAB$^+$86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[DFH$^+$91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The COQ proof assistant user's guide: Version 5.6. Rapports Techniques 134, INRIA, Rocquencourt, France, December 1991.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[EHDM93] *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes.

[FBHL84] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of*

Mathematics. North-Holland, Amsterdam, The Netherlands, second printing, second edition, 1984.

[FGJM85] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[GH93] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[How91] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 162–172, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[How96] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST'96*, pages 85–101. Number 1101 in Lecture Notes in Computer Science, Springer Verlag, 1996.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[LP97] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? SRC Research Report 147, Digital Systems Research Center, Palo Alto, CA, May 1997. Available at `http://www.research.digital.com/SRC`.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 341–386, New York, NY, 1989. Springer-Verlag.

[MMMS90] Albert R. Meyer, John C. Mitchell, Eugenio Moggi, and Richard Statman. Empty types in polymorphic lambda calculus. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming, pages 273–284. Addison-Wesley, 1990.

[OS97] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1997. Revised version of SRI-CSL-93-9. To appear as a NASA Contractor Report.

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998. Three volumes: Language, System, and Prover Reference Manuals.

[RAISE92] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice-Hall International, Hemel Hempstead, UK, 1992.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.