The TYPELAB Specification and Verification Environment *

F.W. von Henke, M. Luther, H. Pfeifer, H. Rueß,
D. Schwier, M. Strecker, M. Wagner

Universität Ulm
D-89069 Ulm, Germany

1 Introduction

TYPELAB is an experimental environment that permits the specification of software and hardware systems in a modular fashion. Modules are first-class objects that can be manipulated in different ways, for example through refinement in a stepwise process. A high degree of abstraction and good reuse properties are achieved by genericity. The specification language of TYPELAB is based on a very expressive type theory, the *Extended Calculus of Constructions* [Luo94], which gives the system a sound semantic foundation. The pure type theory has been augmented by syntactic constructs that permit an intuitive handling of the entities managed by the system. TYPELAB comprises a proof assistant which is primarily thought to be used as an interactive proof checker. However, a sequent-style theorem prover has been developed for automatically solving medium-sized problems in restricted fragments of the logic. During a specification development activity or during a proof, a knowledge base can be consulted. This knowledge base contains previous developments and theorems that have either been added explicitly or which can be inferred by the system by a kind of "inheritance" mechanism.

2 The Specification Language

The TYPELAB specification language (see [vHDR+95] for a more detailed discussion) has properties that make it suitable both for small-scale and large-scale development and verification tasks. The type theory *ECC* on which TYPELAB is based has been enriched by constructs found in traditional specification languages. Thus, although the full expressiveness of the underlying logic and type system is made available to the user, no prior knowledge of type theory is required. Types not only serve the purpose offered by sorts in traditional algebraic specifications; they also permit to express semantic constraints, as for example restrictions on the domain of partial functions, and thus lead to a concise notation.

* This research has partly been supported by the "Deutsche Forschungsgemeinschaft" within the "Schwerpunktprogramm Deduktion"

The language can basically be perceived as a functional programming language with a strong type system. By the use of dependent function types, ML-style polymorphism can easily be expressed. As in the PVS system [OSR93], partial functions can be defined by means of semantic subtypes, which restrict the domain of a function to arguments satisfying certain properties. Semantic subtypes can be coded as dependent record types in the underlying type theory *ECC*. A higher-order logic is tightly integrated in the type system via the type-theoretic "propositions-as-types" paradigm.

Structuring in the large is achieved through a powerful module system which considers specifications (or theories, in mathematical parlance) as first-class objects. Specifications are interpreted as dependent record types of *ECC* and thus can be treated in a uniform manner with other objects of the software development process. In particular, parameterized specifications can be expressed as functions that map a parameter P of a specification type to a specification type depending on P. Similarly, a refinement of specifications can be understood as a function having a specification as its domain and a specification as range. The tight integration of the module system with the type system and the higher-order specification logic yield a language which is more expressive than the languages of most systems and formalisms propagating a style of "parameterized programming" [Gog84].

In recent years, implementations based on type theories have gained recognition, notably Alf [MN94], Coq [Cor95], Lego [LP92] and Nuprl [Con86]. These systems are dedicated very much to the underlying logical formalism, and program derivation is closely linked to a constructive proof of a formula which embodies the specification of the program. As opposed to this view, TYPELAB regards expressive type theories as an adequate foundational formalism, but otherwise tries to abstract from the particular type theory by offering a syntax that makes it similar to advanced verification environments like PVS [OSR93].

In particular, abstract datatypes can be specified in a familiar notation, i.e. by first declaring functions that make up the signature and then specifying axioms the functions have to fulfill. In addition, theorems can be stated in a specification. They give rise to proof obligation that may later be discharged with the aid of the proof assistant.

3 System Support

The TYPELAB system consists of a type checker, a proof assistant and, as an experimental feature, a knowledge base of developments and proofs. These components are closely integrated and are accessible through a uniform user interface.

The type checker ensures the type correctness of the objects manipulated by the system and generates proof obligations, if necessary. Proof obligations arise if a term is coerced to a semantic subtype, if theorems are stated in a specification, and if maps between specifications are given. In the latter case, a map specifies how the functions of an abstract specification can be implemented in a refined specification, without showing that the properties of the functions are preserved. The system then generates proof obligations derived from the appropriately instantiated axioms of the abstract specification.

The proof assistant can be started either with a system-generated proof obligation or with a proof goal issued by the user. Proof construction is to be understood in a large sense: Proofs are not limited to logical statements, but equally well permit the construction of a refinement map between specifications. The proof assistant is thought to be used primarily in an interactive mode, where proof goals can be decomposed by rules of a sequent-style calculus. Some automation is offered by search procedures which are complete for restricted fragments of the logic. Among them are a decision procedure for (intuitionistic) propositional logic and a complete proof procedure for the first-order fragment. Furthermore, the proof assistant provides a tactic language, which permits to define proof procedures for repeatedly occurring patterns of deduction. The proof assistant is further described in [Wag95].

Currently, a knowledge-based component is under construction, which aims at organizing mathematical entities and components of the software development process, such as conceptual vocabulary (for example terminology describing different kinds of orders) or (parameterized) specifications. These objects are arranged in a taxonomy which is structured by a subsumption relation. In the case of a terminology, this is a generalized implication (see [Lut95] for details). In the case of parameterized specifications, the subsumption relation is a covariant refinement relation. The construction of such a database not only has the advantage of making the relation between objects more perspicuous. It also permits to inherit statements made about certain objects in the taxonomy or theorems asserted in a specification, thus promoting reuse of proofs and developments in different contexts.

4 Future Extensions

A research prototype of the TYPELAB system has been implemented and is currently being tested on small examples drawn from the domains of program development and mathematics. Enhancements of the syntax and further automation of the prover are envisaged. They will permit to tackle more substantial examples from hardware and software design.

Currently, work is in progress to integrate specification operators in the style of ASL [Wir86] into the TYPELAB language, for example extension of a specification or renaming of the signature. These operators facilitate the construction of knowledge bases of specifications.

The knowledge-based component of TYPELAB will be further extended to allow not only inheritance of theorems, but also proof abstraction and analysis of applicability conditions of program transformations and refinement steps.

References

[Con86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Cor95] Cristina Cornes et al. *The Coq Proof Assistant Reference Manual.* INRIA Rocquencourt and CNRS-ENS Lyon, 1995.

[Gog84] J.A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.

[LP92] Zhaohui Luo and Robert Pollack. *LEGO Proof Development System: User's Manual.* University of Edinburgh, Department of Computer Science, 1992.

[Luo94] Zhaohui Luo. *Computation and Reasoning.* Oxford University Press, 1994.

[Lut95] Marko Luther. Wissensbasierte Methoden zur Beweisunterstützung in Typentheorie. Master's thesis, Universität Ulm, 1995. Available at URL http://www.informatik.uni-ulm.de/ki/Forschung/Deduktion /ml-diplomarbeit.html.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Springer LNCS*, pages 213–237, 1994.

[OSR93] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language.* Computer Science Lab, SRI International, Menlo Park CA 94025, March 1993.

[vHDR+95] F.W. von Henke, A. Dold, H. Rueß, D. Schwier, and M. Strecker. Construction and deduction methods for the formal development of software. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, Springer LNCS 1009. 1995.

[Wag95] Matthias Wagner. Entwicklung und Implementierung eines Beweisers für konstruktive Logik. Master's thesis, Universität Ulm, 1995. Available at URL http://www.informatik.uni-ulm.de/ki/Forschung /Deduktion/mw-diplomarbeit.html.

[Wir86] Martin Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42:123–249, 1986.

This article was processed using the LaTeX macro package with LLNCS style