# Construction and Deduction Methods for the Formal Development of Software

F. W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker

Abt. Künstliche Intelligenz
Universität Ulm

**Abstract.** In this paper we present an approach towards a framework based on the type theory ECC (Extended Calculus of Constructions) in which specifications, programs and operators for modular development by stepwise refinement can be formally described and reasoned about. We demonstrate how generic software development steps can be expressed as higher-order functions and how proofs about their asserted effects can be carried out in the underlying logical calculus.
For formalizing transformations that require syntactic manipulation of objects, we introduce a two-level system combining a meta-level and an object level and show how to express and reason about transformations that faithfully represent object-level operators.

## 1 Introduction

Modern software engineering regards software development as an evolutionary process [Wir95, BP81]. One view of this process is that, starting from abstract, high-level requirement specifications, a series of refinement or implementation steps is applied to successive levels of specification, eventually yielding a program as the final result of the process. In a more formal context, it must be demonstrated for each step that the refined specification or implementation satisfies the properties postulated by the previous (higher level) specification. Then the final program will satisfy the initial requirements, provided suitable properties of compositionality of steps hold.

Past experience has shown that formal verification of software developments requires more effort and higher costs than can be justified in most situations, making traditional *post mortem* verification rather impractical. As an alternative, we may analyze the development process further and identify certain steps that are applied repeatedly as refinement patterns. If we succeed in formalizing such patterns and verifying their properties, we may considerably reduce the effort required for the formal verification of the development process. In particular, it is desirable to formalize a development pattern as an *operator* that transforms specifications into new specifications and to prove that the result of applying the operator yields, for example, a refinement of the argument specification. Accordingly, the demonstration of correctness for each development step that is an instance of a formalized pattern has been reduced to showing that the operator is applicable.

In this paper we present an approach towards a framework in which we can formally describe and reason about specifications, programs and development operators and apply the method outlined above. Our approach is based on a type theory, the *Extended Calculus of Constructions* (ECC) [Luo90, Luo91a], as the unifying logical foundation. Building on ECC, we define a specification language, TYPELAB; roughly, it introduces syntactic constructs that are closer to the style of algebraic specifications and more readable than the language of the "raw" logic, while its semantics is grounded in the type theory. In essence, a specification represents a type, and a member of that type is a realization of that specification. Obviously, such a notion of types as specifications requires types to convey semantic information; as a consequence, demonstrating that an object has a particular type, i.e. type checking, may involve verifying that it satisfies the semantic properties of the type – which, in general, requires theorem proving.

The language is rich enough for expressing specifications, assertions about specifications, and relations between specifications in a natural way. In particular, many generic development steps can be expressed as higher-order functions, and proofs that they have the asserted effect can be carried out in the underlying logical calculus; a formalization of this kind will be presented in Sect. 4. It seems, however, that in many cases the formalization of development patterns requires a direct description of how the text of a specification has to be modified, for instance for optimizing transformations; thus, such patterns must be formalized as operators on syntactic representations of specifications. Then the verification that applying an operator indeed establishes the asserted relationship between its source and target typically requires relating the *syntactic* manipulation (i.e., how the text of the source specification is modified to yield the text of the target) to the *semantic* relationship between the meanings of those texts. To facilitate this kind of reasoning a two-level formal system has to be provided: the syntax of the object language is represented by data types of the meta-level, and a reflection principle serves to link syntactic structures to their meaning at the object level. In Sect. 4 we develop such a two-level system for TYPELAB and show by means of a simple example how operators can be formalized and reasoned about.

The remainder of the paper is organized as follows. Section 2 contains a brief description of the type theory used. In Sect. 3 we introduce the specification language TYPELAB. Section 4 presents the two main approaches to formalizing development steps: using higher-order functions, and using meta-operators; for the latter approach the two-level formal system is developed. Section 5 discusses aspects of the TYPELAB implementation. The final section contains a brief summary and conclusions.

## 2 Type-theoretic Foundation

The formal basis of our approach is the type theory *Extended Calculus of Constructions* (ECC) [Luo90, Luo91a] augmented by inductive types. We briefly summarize those features of the type theory that are needed in this paper.

ECC, like all advanced type theories, may be regarded as an extension of the (simply typed) lambda calculus [Chu40] by a more powerful type system. In our context, the most important extensions are the addition of *dependent types* and *type universes*.

$\Sigma$-types (strong sum types) generalize Cartesian products: $\Sigma x : A.\, B$ is the type of pairs $(a, b)$ such that $a$ is a member of type $A$ and $b$ is in $B\,[x := a]$.[1] $\Pi$-types (abstraction types) generalize function types. Intuitively, $\Pi x : A.\, B$ is the type of dependent functions with domain $A$ and codomain $B$ where $B$ may depend on the element to which the function is applied.

A *type universe* is a type which has types as its members. ECC offers two kinds of universes, *Prop* and $Type_i$, for natural numbers $i$. By the Curry-Howard principle of *propositions-as-types* [CF58, How80], logical formulas are considered as the types of their proofs. They are included in the universe *Prop* and data types reside in the universes $Type_i$. Coquand and Huet [CH85] demonstrate how logical connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$), logical quantification ($\forall$, $\exists$) and *Leibniz equality* ($a = b$) are coded. Strong sums and type universes in ECC prove to be useful for encoding program specifications and abstract implementations between specifications, and for modular development by stepwise refinement [Luo91b].

The treatment of rules and proofs is based on the notion of *judgement*. Typing judgements are of the form $\Gamma \vdash M : A$ and express the fact that in context $\Gamma$ term $M$ is of type $A$, where a context is defined as a finite sequence of declarations $x : A$. Depending on the situation, $a : A$ may be interpreted as "$a$ is of type $A$", "$a$ is a proof of formula $A$", or "$a$ meets specification $A$". A term $M$ is *well-typed* in context $\Gamma$, if $\Gamma \vdash M : A$ for some $A$. A type $A$ is *inhabited* under context $\Gamma$ if and only if there exists a term $M$ such that $\Gamma \vdash M : A$ is derivable. For a complete presentation of typing rules and a notion of derivability of judgements see [Luo90]. ECC has many good meta-theoretic properties. It obeys the Church-Rosser property, is strongly normalizable, and type checking is decidable.

## 3 Specification in TYPELAB

In the following we extend the calculus ECC by constructs for representing units of the software development process [Wir95]. The design of these constructs is influenced mainly by the PVS [ORS92, ORSv95] specification language and Extended ML [ST89]. The extensions to ECC are quite expressive in the sense that most of the mathematical and computational concepts we wish to describe can be formulated very directly and naturally.

[1] Capital letters and $a$, $b$ denote terms of the term calculus of ECC, while $x$, $y$ denote variables. $N\,[x := M]$ denotes the substitution of a term $M$ for all free occurrences of $x$ in the term $N$.

Type constructors are introduced to form Cartesian products, (dependent) record types, semantic subtypes, and specifications. All these constructs are special forms of strong sum types in ECC; they are, however, handled differently by the typing system and therefore require special syntax. Cartesian products and record types are of the form $A_1 \times \ldots \times A_n$ and $\ll x_1 : A_1, \ldots, x_n : A_n \gg$ respectively; their elements are tuples $(a_1, \ldots, a_n)$. The common dot notation denotes selection of record fields.

A semantic subtype $\{x : A \mid P\}$ includes those members of type $A$ which satisfy predicate $P$. Elements of the semantic subtype are denoted by $a[p]$, where $a$ is a member of type $A$ and $p$ is a proof term of type $P[x := a]$. (This notation is possible because proofs can be expressed as usual terms). A distinctive feature of the typing system is a conversion mechanism that is able to convert members of one type to members of a different type automatically. For example, applying a function that requires a member of $\{x : \mathbb{N} \mid \mathit{Odd}(x)\}$ to the natural number 5 is illegal, because 5 is not a member of the subtype. But if one can find a term $p$ which is a proof of $\mathit{Odd}(5)$, we may rewrite the application using $5[p]$ instead of 5. Since in general it is not possible to find the required proofs automatically, proof obligations are generated. A proof obligation is a placeholder for a term which will be filled in later by the prover. These proof obligations can be postponed because the type checker only requires type information.

A specification consists, as usual, of a signature part and an axiom part; the signature part normally corresponds to a dependent $\Sigma$-type, the axiom part is a collection of propositions (elements of type *Prop*) that restrict the set of acceptable "models" of the signature. For instance, the following specification declares a type *Setoid* as consisting of a type $T$ together with a binary Boolean function $eq$ on $T$ that is restricted to be an equivalence relation:

$$
\begin{array}{ll}
\mathit{Setoid} := & \textsc{spec} \\
& \quad T : \mathit{Type},\ eq : T \times T \to \mathbb{B} \\
& \textsc{with} \\
& \quad Ax :\ \mathit{equivalence}(eq) \\
& \textsc{end}
\end{array}
$$

Realizations of such specifications are structures that satisfy the axioms. For example, the structure $\textsc{struc}\ T := \mathbb{B},\ eq := eq_{\mathbb{B}}\ \textsc{end}$ is of type *Setoid* if the condition $\mathit{equivalence}(eq_{\mathbb{B}})$ holds. Whenever a structure is type checked and no proof terms are given, proof obligations are generated to fill out any missing proofs. The proof obligations are derived from the specification by substituting terms from the structure into the axioms. In this example the obligation is $\mathit{equivalence}(eq_{\mathbb{B}})$. Let $p$ be a proof of this proof obligation, then the structure above is converted into:

$$\textsc{struc}\ T := \mathbb{B},\ eq := eq_{\mathbb{B}}\ \textsc{end}\ [p]\ \ : \mathit{Setoid}$$

The conversion mechanism is also used by the casting construct (::) of TYPELAB. A term $M :: A$ causes the typing system to check if $M$ is a member of type $A$. If the type check fails, the system tries to generate a term $M'$ of type

$A$ from $M$ by introducing proof obligations. This feature is used to generate the proof obligations that are necessary to establish the correctness of the development process. The following function, for example, realizes a refinement map with import specifications $imp_1$ and $imp_2$, and the export specification $exp$.

$$
\begin{array}{l}
\rho := \ \lambda\ r_1 : imp_1,\ r_2 : imp_2. \\
\qquad\qquad \text{STRUC} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad \text{END} \ :: \ exp
\end{array}
$$

Type casting of the function body produces proof obligations that intuitively state: if realization $r_i$, $i = 1, 2$, fulfills the axiom part of specification $imp_i$ then $\rho(r_1, r_2)$ fulfills the axioms of $exp$.

The mechanisms to form *inductive datatypes* follow the extension of ECC by Ore [Ore92]. Polymorphic lists, for example, are defined by

$$List := \ \lambda\ T : Type.\ \text{DATATYPE}\ X : Type.\ nil \ \mid \ cons : T \times X$$

Note that the names of the constructors for inductive datatypes have to be introduced explicitly (e.g. $mkNil := \ \lambda\ T : Type.\ \text{INTRO}(List(T), nil)$). The CASE construct is a generic construct for inductive datatypes; it allows both for structural induction over inductively defined datatypes and for the definition of functions by means of (higher–order) primitive recursion; it can be seen as a variant of the concept of *hom–functionals* [vH76] and exhibits the natural correspondence between the structure of a program (or proof) and the data structure. The specific inductive structure to which CASE is being applied is determined by the type of the first argument. For example, the function *map* on polymorphic lists,[2]

$$
\begin{array}{l}
map := \ \lambda\ T, S \mid Type, l : List(T), f : T \to S. \\
\qquad\qquad \text{CASE}\ l\ \text{OF} \\
\qquad\qquad\quad nil : mkNil(S), \\
\qquad\qquad\quad cons :\ \lambda\ (t, l_1) : T \times List(T),\ y : List(S). \\
\qquad\qquad\qquad\qquad mkCons(f(t), y) \\
\qquad\qquad \text{END}
\end{array}
$$

is completely specified by describing its behavior for each of the constructors separately. In the second case of the CASE construct ($cons$), the result of applying the function $f$ to the head element $t$ is concatenated to the result $y$ of what effectively is the recursive call of *map*. Inductive datatypes representing Booleans ($\mathbb{B}$), natural numbers ($\mathbb{N}$) and polymorphic lists ($List$) together with appropriate operators are predefined.

The FIX construct allows for defining recursive functions in a restricted form: mutual recursion is not allowed, and functions must be proven to be total.

[2] The notation $\lambda\ T \mid Type.\ \ldots$ is used to denote type parameters which usually are not provided explicitly, i.e. are left implicit and deduced by type checking.

Consider, for example, the definition of the factorial function:

$$\begin{array}{l} \text{FIX } f : \mathbb{N} \to \mathbb{N}. \\ \quad \lambda\, n : \mathbb{N}.\ \text{IF } \mathit{isZero}(n) \text{ THEN } 1 \text{ ELSE } n * f(n-1) \text{ END} \\ \text{MEASURE } \lambda\, x : \mathbb{N}.\ x \end{array}$$

The function following the keyword MEASURE is required for demonstrating termination of the function being defined recursively by means of the FIX construct, following the approach of PVS. It has the same domain as the recursively defined function and, in this case, range type $\mathbb{N}$. The definition generates the *termination correctness condition*

$$\forall\, n : \mathbb{N}.\ \mathit{isZero}(n) \neq \mathit{true} \ \Rightarrow\ n - 1 <_{\mathbb{N}} n$$

using the standard ordering $<_{\mathbb{N}}$ on $\mathbb{N}$ as default. This condition must be discharged to ensure well-typedness of $f$. Measure functions can also be utilized in the obvious way to prove properties about recursive functions by means of *Noetherian induction*.

## 4 Formalizing Development Steps

In this section we present two approaches to formally representing and reasoning about software development steps in TYPELAB:

- by higher-order functions,
- by meta-functions.

### 4.1 Representation of Steps by Higher-Order Functions

The formalization of transformations using higher-order patterns has been considered by several researchers. In [HL78], for example, program transformations for recursion removal are expressed as second-order patterns defined in the simply typed $\lambda$-calculus [Chu40]. In contrast to this treatment we use the powerful framework of TYPELAB and demonstrate how it is possible to formalize and verify a "large" development step ; this is illustrated by a schematic algorithm *global-search*. Due to space limitations, only the most essential features can be sketched, the rigorous mathematical treatment and verification is presented in [Dol94]. To a large extent, this work follows the approach developed by D. Smith [Smi87]. Independently of the work described here, a formal treatment of some of the "larger" steps has also been carried out by C. Kreitz [Kre93] in the context of NUPRL [Con86].

*Global-search* is a generalization of well-known search strategies such as *backtracking* and *depth-first-search* [Smi87]. The basic idea of global-search is to represent and manipulate sets of candidate solutions. Starting from an *initial* set containing all solutions, a global-search algorithm repeatedly *extracts* solutions, *splits* sets into subsets until no sets remain to be split. Sets are represented implicitly by *descriptors*; valid, i.e. meaningful, descriptors are characterized by a predicate $J$. A predicate *satisfies* on descriptors determines whether a candidate solution is in the set denoted by the descriptor. The whole process can be regarded as a search procedure on trees in which nodes represent sets implicitly described by the type $S$ of set descriptors and arcs represent the *split* operation.

Starting from a requirement specification, an extension of this specification defines the additional datatypes and operations needed to realize a global-search algorithm. This extended structure is expressed in a specification called *global_search_theory*. Based on this theory an abstract generic algorithm can be defined. Instantiating the abstract scheme with the specific problem structure together with a proof that the structure satisfies the axioms of *global_search_theory* suffices to synthesize an algorithm realizing a constructive solution of the problem. Using this method, we have derived a *key-search* algorithm and shown that its verification is easily obtained by applying the correctness proof of the transformation to the specific problem structure [Dol94].

One starts with the following specification:

$$Problemspec := \ll D : Type, R : Type, I : D \to Prop, O : D \times R \to Prop \gg$$

where $D$ is the domain type, $R$ the range type, $I$ the input condition restricting $D$ to legal inputs and $O$ the input/output relation. The problem is then formally described by[3]

$$\begin{aligned} req\_spec := \ & \lambda\,(D, R, I, O) : Problemspec. \\ & \forall\, x : D.\ I(x) \Rightarrow \exists\, S : Set(R).\ \forall\, elem : R. \\ & ((elem \in S) = true) \Leftrightarrow O(x, elem)) \end{aligned}$$

The schematic algorithm defined below realizes a constructive proof of this (parameterized) proposition.

The theory *global_search_theory* is given as a specification parameterized by an object of type *Problemspec*. The theory is sketched in Fig.1. The following properties (in the theory expressed formally as axioms) must hold for the theory components:

1. The initial descriptor $init(x)$ is a valid descriptor.
2. If $r$ is a legal descriptor then all its (immediate) descendants $s$ calculated by the *split* operation are legal descriptors.
3. All solutions must be contained in the set described by $init(x)$.
4. A candidate solution $z$ is in a set described by $s$ if and only if it can be extracted from $s$ or one of its descendants.
5. Elements $z$ which can be extracted from a set $r$ (i.e. $extract?(z, r) = true$) are contained in the set $extract(x, r)$ if they satisfy condition $O$.
6. There are no loops w.r.t. the transitive closure of the relation *split?*.
7. Every legal descriptor has at most one predecessor.

The function $F_{gs}$ (Fig.2) defines the schematic algorithm which takes as input a realization of a *global_search_theory* and two additional functions *arbsplit* for

$$
\begin{array}{l}
\mathit{global\_search\_theory} := \ \lambda\ (D, R, I, O) : \mathit{Problemspec}. \\
\quad \text{SPEC} \\
\qquad S : \mathit{Type} \\
\qquad J : D \times S \to \mathit{Prop} \\
\qquad \mathit{init} : D \to S \\
\qquad \mathit{satisfies} : R \times S \to \mathit{Prop} \\
\qquad \mathit{split?} : D \to (S \times S \to \mathit{Prop}) \\
\qquad \mathit{split} : D \times S \to \mathit{Set}(S) \\
\qquad \mathit{extract?} : R \times S \to \mathit{Prop} \\
\qquad \mathit{extract} : D \times S \to \mathit{Set}(R) \\
\quad \text{WITH} \\
\qquad \mathit{ax1} : \forall\, x : D.\ I(x) \Rightarrow J(x, \mathit{init}(x)), \\
\qquad \mathit{ax2} : \forall\, x : D,\ r, s : S.\ (I(x) \wedge J(x, r) \wedge \mathit{split?}(x)(r, s)) \Rightarrow J(x, s), \\
\qquad \mathit{ax3} : \forall\, x : D, z : R.\ (I(x) \wedge O(x, z)) \Rightarrow \mathit{satisfies}(z, \mathit{init}(x)), \\
\qquad \ldots \\
\quad \text{END}
\end{array}
$$

**Fig. 1.** Definition of a global_search_theory

selecting an arbitrary element of a set and *tcl* which is used for termination. The result is then a function $f$ defined on a set of descriptors, a set of solutions, and a legal input $x$. It selects at each step a descriptor from the active set, computes its descendants, extracts solutions and repeats this operation on each subset until all nodes have been considered. The initial value of the active set is given by $init(x)$, and the set of solutions is empty. The function *tcl* produces for a given set of nodes in the search tree its (finite) set of successors with respect to the relation *split?*, i.e. it calculates the transitive closure of *split?*. This specifies a finite depth of the search tree. One implicitly obtains a finite width by using the polymorphic type *Set* of finite sets, i.e. *split* produces for a given node the finite set of its (direct) descendants. To guarantee well-typedness of the recursive function we must supply a *measure* function. Here we use the cardinality of the transitive closure of the active set of nodes. The concept of semantic subtypes is used to represent an *invariant* constraining the domain *F_Type* of $f$. The predicate *Invar* ensures that

1. every node of the active set is a legal descriptor;
2. all elements of the set *solution* fulfill condition $O$;
3. for two arbitrary nodes $s_1, s_2$ of the active set, $s_2$ is not a successor of $s_1$ w.r.t. the relation *split?*.

To establish the correctness of the defined development step one has to show that for an arbitrary problem specification and global search theory the instantiated function $f$ is indeed a constructive solution, i.e. $f$ calculates the set of

[3] We suppose that the type $Set(T)$ of finite sets over a type $T$ together with suitable operations is given.

$$
\begin{array}{l}
F_{gs} := \lambda\,(D,R,I,O) : \mathit{Problemspec},\ gs : \mathit{global\_search\_theory}((D,R,I,O)),\\
\qquad \mathit{arbsplit} : \ldots,\ \mathit{tcl} : \ldots.\\
\qquad \text{LET}\\
\qquad\quad F\_Type := \ll \mathit{active} : \mathit{Set}(gs.S), \mathit{solution} : \mathit{Set}(R),\\
\qquad\qquad\quad x : \{D \mid \mathit{Invar}((D,R,I,O), gs, \mathit{active}, \mathit{solution}, x)\} \gg\\
\qquad \text{IN}\\
\qquad\quad \text{FIX}\ f : F\_Type \rightarrow \mathit{Set}(R).\ \ \lambda\,(\mathit{active}, \mathit{solution}, x) : F\_Type.\\
\qquad\qquad \text{IF}\ \mathit{empty?}(\mathit{active})\ \text{THEN}\ \mathit{solution}\\
\qquad\qquad\quad \text{ELSE}\\
\qquad\qquad\quad \text{LET}\\
\qquad\qquad\qquad (r, A_1) := \mathit{arbsplit}(\mathit{active}),\\
\qquad\qquad\qquad \mathit{Newactive} := A_1 \cup gs.\mathit{split}(x,r),\\
\qquad\qquad\qquad \mathit{Newsolution} := \mathit{solution} \cup gs.\mathit{extract}(x,r)\\
\qquad\qquad\quad \text{IN}\\
\qquad\qquad\qquad f(\mathit{Newactive}, \mathit{Newsolution}, x)\\
\qquad\quad \text{MEASURE}\\
\qquad\qquad \lambda\,(\mathit{active}, \mathit{solution}, x) : F\_Type.\ \mathit{card}(\mathit{tcl}(\mathit{active}, \mathit{solution}, x))
\end{array}
$$

**Fig. 2.** The schematic algorithm *Global Search*

all elements of the range type $R$ which satisfy the condition $O$. The soundness theorem is given in Fig.3. Additionally, to ensure type correctness some type correctness conditions are generated. The first one states that the measure function applied to the parameters of the recursive call yields a smaller value than the function called with the original parameters. Furthermore, the parameters of the recursive call and the initial parameters must satisfy the invariant of $f$. All proof obligations have successfully been discharged using the (interactive) higher-order Gentzen prover of the PVS specification system [ORS92].

The techniques outlined above can readily be used to formalize many ge-

$$
\begin{array}{l}
\mathit{Soundness\_Theorem} :=\\
\quad \forall\,(D,R,I,O) : \mathit{Problemspec},\ gs : \mathit{global\_search\_theory}((D,R,I,O)),\\
\qquad \mathit{arbsplit} : \ldots,\ \mathit{tcl} : \ldots,\ x : \{D \mid I(x)\},\ y : R.\\
\qquad \text{LET}\\
\qquad\quad F\_inst := F_{gs}((D,R,I,O), gs, \mathit{arbsplit}, \mathit{tcl}),\\
\qquad\quad \mathit{init\_set} := \mathit{insert}(gs.\mathit{init}(x), \varnothing_{gs.S}),\\
\qquad\quad \mathit{init\_sol} := \varnothing_R,\\
\qquad\quad \mathit{sol\_set} := F\_inst(\mathit{init\_set}, \mathit{init\_sol}, x)\\
\qquad \text{IN}\\
\qquad\quad (y \in \mathit{sol\_set} = \mathit{true}) \Leftrightarrow O(x,y)
\end{array}
$$

**Fig. 3.** Soundness Theorem of *Global Search*

neric development steps including transformations such as *divide-and-conquer*, *dynamic programming* and those investigated by the Munich CIP group [CIP87, Par90].

### 4.2 Meta-Operators

Many typical development steps are not representable with the language constructs introduced in Sect. 3. Consider, for example, the simple task of replacing a certain axiom $P_i$ in a specification text by another axiom $Q$. If $Q$ implies $P_i$ then one can construct a refinement map from the modified specification to the original one. More precisely: let $\Gamma$ be the current context, abbreviate $x_1 : A_1, \ldots, x_n : A_n$ by $\mathbf{x} : \mathbf{A}$, and define a specification

$$sp_1 := \text{SPEC } \mathbf{x} : \mathbf{A} \text{ WITH } p_1 : P_1, \ldots,\ p_i : P_i, \ldots,\ p_m : P_m \text{ END}$$

that is well-typed in $\Gamma$. Furthermore, assume that the judgement $\Gamma, \mathbf{x} : \mathbf{A} \vdash p : Q \Rightarrow P_i$ is derivable.[4] It is our task to construct a realization of $sp_1$ relative to a realization of specification

$$sp_2 := \text{SPEC } \mathbf{x} : \mathbf{A} \text{ WITH } p_1 : P_1, \ldots,\ q : Q, \ldots,\ p_m : P_m \text{ END}$$

A refinement map from specification $sp_2$ to specification $sp_1$ is constructed as

$$\begin{aligned}\rho := {} & \lambda\, r : sp_2. \\ & \quad \text{STRUC } x_1 := r.x_1, \ldots, x_n := r.x_n \text{ END } [r.p_1, \ldots, p(r.q), \ldots, r.p_m]\end{aligned}$$

and the type introduction rule for structures immediately yields:

$$\Gamma \vdash \rho : sp_2 \rightarrow sp_1$$

A transformation of this kind which takes a specification $sp_1$, a formula $Q$, and an index $i$ and results in a new specification $sp_2$ by replacing the $i$-th axiom in $sp_1$ by $Q$ needs both access to internal structure in order to manipulate syntactical text and the correctness of this formalization involves reasoning about derivability of judgements, i. e. meta-reasoning. Furthermore, this development step deals with a term $Q$ that is not necessarily well-typed in the current context $\Gamma$ but only in $\Gamma, \mathbf{x} : \mathbf{A}$.

In the following we describe a *meta-architecture* that allows one to express such development steps and transformations by means of functions on representations of programs (proofs) and specification texts. These functions are called *meta-functions* and are amenable to formal treatment; e.g. one can state and prove characteristic properties about them.

Historically, meta-architectures were first formalized and investigated by logicians, where the pioneering work has been carried out by Gödel [Göd31]. From a more application oriented view, meta-level architectures have been used extensively in the realm of mechanical theorem proving [BM81, ACHA90, How88, KC86], since in many cases it is quite straightforward to construct a proof by

[4] Note that Q need not be well-typed in context $\Gamma$ if some $x_i$ occurs free in $Q$.

means of syntactic analysis of the problem at hand [Wey80, AW80]. Here, the important issue is how meta-programming and meta-reasoning can be used to represent software development steps together with expressing a certain semantics of these steps.

In a first step one encodes syntactic categories and the proof theory of TYPELAB within itself following the approach of Gödel. This encoding constitutes the *meta-level*. On this encoding one can write (almost) arbitrary functions and express relations like "$x$ is a free variable in $M$" or "the result of substituting the term $N$ for all free occurrences of the variable $x$ in $M$ yields $L$". A particularly important predicate is the derivability predicate expressing the relation that "$M$ is of type $A$ in context $\Gamma$". These features allow to encode development steps (proof steps) by meta-functions, and to express and prove "semantic" relations between arguments and results. The adequacy and faithfulness of the encoding yield *reflection principles* that allow one to exchange results between the meta-level and the object level in a sound way.

Due to lack of space we can merely present a fragmentary sketch of the architecture. A detailed treatment can be found in [Rue95, Pfe95]. One first represents syntactical categories of the object language syntax by means of the inductive datatype *AbsTrm*. The elements of this data type can be seen as abstract syntax of terms. This abstract syntax does not necessarily represent well-typed terms. Representations of specifications, for example, can be formed by means of the constructor *mkSpec* of type $List(Id \times AbsTrm) \times List(Id \times AbsTrm) \rightarrow AbsTrm$. The first argument represents the signature, while the second one represents the axiom part; *Id* is just the type for identifiers. It is straightforward to introduce recognizers and selectors for each alternative in the datatype *AbsTrm*. For specifications we have the recognizer *isSpec* and selectors *specSig* and *specAxms*. Recognizer $isSpec(M)$ yields *true* if and only if $M$ represents a specification, while *specSig* and *specAxms* respectively select the (representations of the) signature and the axiom part. In the following we also utilize the constructor *mkStruc* with corresponding selectors *strucDefs* and *strucPrfs*.

Contexts are represented by elements of type *Ctxt* which is a list of (representations of) type assignments $x : A$ while judgements are represented by elements of $Jdgmt := Ctxt \times AbsTrm \times AbsTrm$. The data types *AbsTrm*, *Ctxt*, and *Jdgmt* are called *representation types* and elements of them are meta-terms.

A quoting mechanism '.' associates syntactic categories of the object level like terms, contexts, and judgements with meta-terms; for example:

$$\begin{aligned}&\text{' SPEC } x_1 : A_1, x_2 : A_2 \text{ WITH } p_1 : P_1, p_2 : P_2 \text{ END '} := \\ &\quad mkSpec(\langle(\,\text{'}x_1\text{'}\,,\,\text{'}A_1\text{'}\,),(\,\text{'}x_2\text{'}\,,\,\text{'}A_2\text{'}\,)\rangle,\ \langle(\,\text{'}p_1\text{'}\,,\,\text{'}P_1\text{'}\,),(\,\text{'}p_2\text{'}\,,\,\text{'}P_2\text{'}\,)\rangle)\end{aligned}$$

Through the mapping '.' object-level constructs become available for discourse at the meta-level.

It is a standard exercise to encode the term calculus. One defines functions *occurs* of type $AbsTrm \times Var \rightarrow \mathbb{B}$ and *substVar* of type $AbsTrm \times Var \times AbsTrm \rightarrow AbsTrm$ by means of higher–order primitive recursion such that $occurs(\text{'}M\text{'}, \text{'}x\text{'})$ reduces to *true* if and only if $x$ occurs free in $M$ and

$subst Var($ '$M$' , '$x$' , '$N$' $)$ reduces to '$M\,[x := N]$' . Binary relations on terms like syntactic equality (modulo *alpha*-convertibility) and convertibility can be coded in a type-theoretic setting by closures of the appropriate binary relations. Likewise derivability of a judgement, denoted by $deriv(.)$, is encoded as the least set (one-place predicate) closed under the rules of the type calculus of TYPELAB. The following fact expresses adequacy and faithfulness of this encoding of derivability

> $\Gamma \vdash M : A$ is derivable if and only if there exists a term $p$ such that $\vdash p : deriv($ '$\Gamma$' , '$M$' , '$A$' $)$

Obviously, a proof of this can neither be carried out at the object level nor at the meta-level, but is rather accomplished in the (informal) theory that allows one to reason about both of these levels. The result above allows one to deduce from the derivability of $\Gamma \vdash M : A$ at the object level the existence of a term of type $deriv($ '$\Gamma$' , '$M$' , '$A$' $)$. This transition from object level to meta-level is named *reflection upwards* while the corresponding change from meta-level to object level is called *reflection downwards* [GS89]. These reflection rules are admissible inferences, and thus, in principle, dispensable. From a practical point of view, however, reflection rules are crucial since they allow to exchange results between object level and meta-level as exemplified in the following.

In the remaining we formalize the development step described in the beginning of this section within our meta-architecture and demonstrate how to apply meta-functions and corresponding correctness results. The meta-function *replaceAxInSpec* replaces in (the representation of) a specification *sp* the (representation of the) $i$-th axiom by (the representation of) another term *axm*, where *replace* is the replacement on lists:

$$
\begin{aligned}
& replaceAxInSpec := \\
& \quad \lambda\, sp : \{AbsTrm \mid isSpec(sp) = true\}, i : Nat, axm : AbsTrm. \\
& \qquad mkSpec(specSig(sp), replace(specAxms(sp), i, axm))
\end{aligned}
$$

It simply replaces the $i$-th element in the list of axiom representations with the argument *axm*. The following predicate states that the resulting (representation of a) specification is indeed a refinement of the argument (representation of a) specification

$$
\begin{aligned}
& \forall\, ctxt : Ctxt, sp : \{AbsTrm \mid isSpec(sp) = true\}, i : Nat, axm, M : AbsTrm. \\
& deriv(append(ctxt, specSig(sp)), M, mkImpl(axm, nth(i, specAxms(sp)))) \\
& \quad \Rightarrow \textsc{let}\ Res := replaceAxInSpec(sp, i, axm), \\
& \qquad\quad N := mkLambda((\ \text{'}r\text{'}\ , Res), \\
& \qquad\qquad\qquad mkStruc(strucDefs(mkRef(\ \text{'}r\text{'}\ )), \\
& \qquad\qquad\qquad\quad replace(strucPrfs(mkRef(\ \text{'}r\text{'}\ )), i, \\
& \qquad\qquad\qquad\qquad mkApp(M, mkProj(mkRef(\ \text{'}r\text{'}\ ), i))))) \\
& \qquad \textsc{in}\ deriv(ctxt, N, mkImpl(Res, sp)) \quad ,
\end{aligned}
$$

where $mkImpl($ '$A$' , '$B$' $)$ is the representation of $A \to B$, and the term $N$ is, despite the ugliness of abstract syntax, a mere formalization of the refinement term constructed in the beginning of this (meta-) exposition. The functions *append* and *nth* denote concatenation of lists and selection of the $n$-th element from a list, respectively. The proof of this correctness result is straightforward and a direct formalization of the informal exposition above; call the corresponding proof $correct_{prf}$. This proof and the reflection principles can be utilized to construct a refinement map between the specification $sp$ and the result of the transformation *replaceAxInSpec*.

Let's go back to our running example and apply *replaceAxInSpec* together with its corresponding correctness result. Again we assume a certain context $\Gamma$ and a specification $sp$. Furthermore, let $'Q'$ be the representation of a certain axiom and $i$ be a fixed natural number. In order to apply $correct_{prf}$ one has to construct an element $'M'$ such that

$$\begin{array}{l} deriv(append('\Gamma', specSig('sp')), \\ \qquad 'M', mkImpl('Q', nth(i, specAxms('sp')))) \end{array}$$

holds. This construction can, of course, be done completely within the meta-level. In many situations, however, it is more appropriate to prove the corresponding problem at the object level; i.e. one has to find a term $M$ such that $M : Q \Rightarrow P_i$ is derivable in context $\Gamma, \mathbf{x} : \mathbf{A}$. The resulting judgement is reflected upwards, yielding a proof $p$ of the predicate above. A simple instantiation of $correct_{prf}$ gives:

$$\begin{array}{l} \vdash correct_{prf}('\Gamma', 'sp', i, 'Q', 'M', p) \\ \qquad : \text{LET } Res := replaceAxInSpec('sp', i, 'Q'),\ N := \ldots \\ \qquad\ \ \text{IN } deriv('\Gamma', N, mkImpl(Res, 'sp')) \end{array}$$

This judgement, finally, is reflected down to the object level in order to get the result that the resulting specification $Res$ indeed is a refinement of the argument specification. Moreover, downward reflection explicitly constructs the object-level refinement map.

As demonstrated above, we are able to formalize conclusions about the object calculus by means of a meta-architecture. This allows one to encode formal development steps *once and for all*; applications of such steps are instances of some meta-level argument, while, in the case of pure object-level reasoning, one has to carry out the same kind of tedious development over and over for each instance of a given problem. Software development systems incorporating a meta-architecture allow users of such systems to add new development (proof) steps only in a sound way. The importance of such features lies in the fact that it is unrealistic to incorporate each conceivable development step in a general–purpose development system. Finally note that, in our approach, meta-functions and meta-properties are essentially the same as object functions and object properties; they only differ in the data types they operate on. Thus, encoding, specification, and proof methods apply to both object-level and meta-level entities.

## 5 Some Notes on the TYPELAB Implementation

An interactive support system for experiments with TYPELAB is under development. The system implements a parser, type checker and pretty printer for the TYPELAB language and provides an interactive proof assistant. The heart of the system is the type checker. It is mainly built around an evaluation function for *pre-terms*. A pre-term is a syntactically correct term that may be ill-typed. The evaluation function takes a pre-term and a set of definitions and, if possible, converts the pre-term to a well-typed term; see also Sect. 3.

In ECC all types belong to exactly one type universe. However, in most cases the specific universe to which a term belongs is irrelevant. For this reason the system offers the possibility to use the anonymous universe *Type* instead of $Type_i$ for a given level $i$. The system tries then to exactly determine the universe level $i$ by maintaining a set of inequalities and checking for consistency [HP89].

Parametric polymorphism is handled by unification. Although higher-order unification is undecidable, most problems which arise in practice from type-checking of polymorphic functions can be solved correctly by the implemented unification algorithm. This result is obtained by coding the universe polymorphism, reductions, alpha convertibility and other features into the unification algorithm.

In an interactive top-down program development process it is desirable to perform typechecking of specifications and their realizations before the development is complete. To achieve this goal, incomplete terms containing placeholders together with suitable type information may be used. Later in the development process, these placeholders will be replaced by members of the appropriate type. This feature, together with a refinement editor, supports a refinement process similar to the one described for Extended ML [ST89, ST92].

For discharging proof obligations arising during typechecking or in the course of a formal development, an interactive proof assistant can be invoked. It has been designed to solve easy problems automatically while leaving the control of major steps to the user. A detailed description of this component of TYPELAB can be found in [Wag95].

## 6 Conclusions and Future Work

In this paper we have presented an approach to formal specification and software development based on type theory. We have discussed the logical basis and illustrated the elementary principles by means of simple examples. Our experience gained so far with the approach supports our hypothesis that specification based on type theory is a viable alternative to the more common algebraic specifications and that many, if not most, interesting operations on, and relationships among, development units can be dealt with by a combination of object-level and meta-level formalization.

The work described here is part of an ongoing investigation into formal methods for software development and effort to develop a suitable framework. Specifically, we plan to develop a basic set of generic algorithms and meta-operators representing development steps, with the long-term goal of compiling some sort of reusable "knowledge base" of programming techniques, and to test whether this approach can be made practical by attacking non-trivial software problems.

## References


[ACHA90] S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The semantics of reflected proof. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE CS Press, 1990.

[AW80] L. Aiello and R.W. Weyhrauch. Using meta-theoretic reasoning to do algebra. In W. Bibel and R. Kowalksi, editors, *5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 1–13. Springer, 1980.

[BM81] R.S. Boyer and J.S. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, chapter 3. Academic Press, 1981.

[BP81] M. Broy and P. Pepper. Programming as a formal activity. *IEEE Trans. on Software Engineering*, 7(1):10–22, 1981.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland Publishing Company, 1958.

[CH85] T. Coquand and G. Huet. Constructions: a Higher-Order Proof System for Mechanizing Mathematics. In B. Buchberger, editor, *EUROCAL'85: European Conference on Computer Algebra*, volume 203 of *LNCS*, pages 151–184. Springer, 1985.

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

[CIP87] CIP System Group. *The Munich Project CIP - Volume II.* volume 292 of *LNCS*. Springer, 1987.

[Con86] R. L. Constable et al. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[Dol94] A. Dold. Formalisierung schematischer Algorithmen. Ulmer Informatik-Berichte 94-10, Universität Ulm, January 1994.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *I. Monatsh. Math. Phys.*, 38:173–198, 1931.

[GS89] F. Giunchiglia and A. Smaill. Reflection in Constructive and Non-Constructive Automated Reasoning. In *Meta-Programming in Logic Programming*, chapter 6, pages 123–140. The MIT Press, 1989.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order-patterns. *Acta Informatica*, 11:31–55, 1978.

[How80] W.A. Howard. The Formulae-as-Types Notion of Construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[How88] D.J. Howe. Computational metatheory in Nuprl. In *Proc. 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 238–257. Springer, 1988.

[HP89] R. Harper and R. Pollack. Type checking, universal polymorphism, and type ambiguity in the Calculus of Constructions. In *TAPSOFT'89, volume II*, volume 310 of *LNCS*, pages 240–256. Springer, 1989.

[KC86] T.B. Knoblock and R.L. Constable. Formalized metareasoning in type theory. In *Proceedings of LICS*, pages 237–248. IEEE, 1986. Also available as technical report TR 86-742, Department of Computer Science, Cornell University.

[Kre93] C. Kreitz. Metasynthesis - deriving programs that develop programs. Technical Report AIDA-93-03, Fachgebiet Intellektik, Technische Hochschule Darmstadt, 1993.

[Luo90] Z. Luo. An Extended Calculus of Constructions. Technical Report CST-65-90, University of Edinburgh, July 1990.

[Luo91a] Z. Luo. A Higher-Order Calculus and Theory Abstraction. *Information and Computation*, 90:107–137, 1991.

[Luo91b] Z. Luo. Program Specification and Data Refinement in Type Theory. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, volume I*, volume 494 of *LNCS*, pages 143–168. Springer, 1991.

[Ore92] Ch.E. Ore. The extended calculus of constructions (ECC) with inductive types. *Information and Computation*, 99:231–264, 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th Intern. Conf. on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[ORSv95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Engineering*, 21(2):107–125, February 1995.

[Par90] H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Pfe95] H. Pfeifer. Eine reflexive Architektur zur Darstellung von Beweis- und SW-Entwicklungsschritten in Typtheorie. Master's thesis, Universität Ulm, 1995.

[Rue95] H. Rueß. *Formal Meta-Programming in the Calculus of Constructions.* PhD thesis, Universität Ulm, 1995.

[Smi87] D. R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, 1987.

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. In *Proc. TAPSOFT 89*, volume 352 of *LNCS*, pages 375–389. Springer, 1989.

[ST92] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: model-theoretic foundations. In *Proc. Intl. Colloq. on Automata, Languages and Programming*, volume 623 of *LNCS*, pages 656–671. Springer, 1992.

[vH76] F. W. von Henke. An algebraic approach to data types, program verification, and program synthesis. In *Mathematical Foundations of Computer Science, Proceedings*, volume 45 of LNCS. Springer, 1976.

[Wag95] M. Wagner. Entwicklung und Implementierung eines Beweisers für konstruktive Logik. Master's thesis, Universität Ulm, 1995.

[Wey80] R. W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1):133–170, 1980.

[Wir95] M. Wirsing et al. A Method for the Development of Correct Software. 1995. In this volume.