6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Type Systems and Program Verification*

Zoltán Csörnyei

Department of Programming Languages and Compilers,
Eötvös Loránd University, Hungary
e-mail: csz@inf.elte.hu

Abstract

A famous slogan by Robin Milner says that "well-typed programs do not go wrong". This slogan essentially asserts the soundness of the type system of programming languages. The question is whether the type system allows us to write meaningful and error-free programs.

Proof generation capabilities of proof construction systems are based on type theory. The base of the theory is the typed λ-calculus. Higher-order type system of higher-order subtyping, known as $F^{\omega}_{\leq}$, has been used as a core calculus for typed languages.

The Curry-Howard isomorphism is a correspondence between type systems and intuitionistic logic: "types are formulas, and expressions are proofs". Types correspond to formulas, and the term "E of type T" correspond to a proof of the formula T, where E is a representation, or encoding, of the proof. For instance, minimal propositional logic corresponds to simply typed λ-calculus, first-order logic corresponds to dependent types, second-order logic corresponds to polymorphic types, etc.

Program verification deals with the question whether a triple $\{Pre\}P\{Post\}$ is consistent. This can be formally defined as $\forall s.(Pre \rightarrow wp(P, Post))$. Type systems allow us to express program properties that are automatically verified.

Techniques for formally specifying, understanding and verifying program behaviors are available, however, the program verification is very expensive. Type systems for program languages are well studied, and there are efforts to refine type systems to allow rich classes of program properties to be expressed and to combine ideas of type theories, verification and interpretation.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification - *Formal methods*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical logic - *Lambda calculus and related systems*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical logic - *Proof theory*;

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742.

Key Words and Phrases: λ-calculus, $F^{\omega}_{\leq}$, proof theory, type system, verification

1. Introduction

Type theory is a basic formalism and *intuitionistic logic* can be encoded inside this theory using the idea of Curry-Howard isomorphism. Due to this *constructive* character, type theory can be viewed as a pure functional programming language and then verifying correctness is reduced to type checking.

The most prominent implementations of type theory are:

- ALF (Chalmers in Göteborg; based on Martin-Löf's extensional type theory)
- Coq (Inria; based on Calculus of Constructions)
- Isabelle/HOL (based on a predicate calculus extended with terms from the simply typed λ-calculus)
- LEGO (Edinburgh; implementing Logical Framework, Calculus of Constructions and Theory of Dependent Types)
- NuPrl (Cornell University; another implementation of Martin-Löf's extensional type theory)
- PVS (based on an extension of simple type theory, namely on the higher-order logic with dependent types)
- Sparkle (Katholieke Universiteit Nijmegen, Holland; a theorem prover for the functional programming language Clean. It can be used to prove the partial correctness of Clean applications. On a smaller scale, it can also be used to prove useful properties of smaller parts of programs.)

All these systems are used in the verification of both hardware and software, the formalization of mathematical proofs, and as tools of research.

Type theory is intended as a theory for program constructions. Programmers write programs and, by using type theory, they prove that their programmes satisfy the specification. This is called *program verification*. Another more sophisticated method is deriving a program from the specification, this is the *program derivation*. Type systems support both methods and the programmer *bridges the gap* between the specification and the program.

In this process there are two different stages and two languages:

- specification process – specification languages
- programming process – programming languages

Specification is expressed *as a set*, the set of all correct programs satisfying the specification, and programming means that the programmer constructs an *element* in the set. Programs are expressed in a *functional programming language*, and the type system is used for deducing the correctness of the program. As a consequence of these principles the type system (*typed λ-calculus*) can be used as programming language, a specification language and a programming logic, as well.

2. Type systems

Let $\mathcal{T} \equiv (\mathcal{S}, \mathcal{I}, \mathcal{R})$ be a *type system*, where $\mathcal{S}$ is the syntax of expressions and types, $\mathcal{I}$ is for judgements, and $\mathcal{R}$ is the set of rules. Type systems fit into the general framework of *formal proof systems*.

The most well-known and well studied type systems are

F_1	*simple typed λ-calculus*,
F_2	*second order (polymorphic) typed λ-calculus*,
$F^{\omega}_{\leq}$	*higher order typed λ-calculus with subtype*.

There are many extensions for type systems for various purposes, for example, existential type was introduced to study abstract data types, and recursive types were introduced to describe recursive data structures, respectively.

Dependent type was introduced for types that depend on expressions, this type is the basis of the Calculus of Constructions *(CC)* and the Martin-Löf type theory.

2.1. Type system à la Church F_1

The *syntax* $\mathcal{S}$ of the simple typed λ-calculus F_1 is defined as follows.

$$
\begin{array}{lcl}
\langle type \rangle & ::= & \langle base\ type \rangle \\
 & | & (\langle type \rangle \ \rightarrow \ \langle type \rangle) \\
\langle \lambda\text{-}expr \rangle & ::= & \langle variable \rangle \\
 & | & (\lambda\ \langle variable \rangle : \langle type \rangle\ .\ \langle \lambda\text{-}expr \rangle) \\
 & | & (\langle \lambda\text{-}expr \rangle\ \langle \lambda\text{-}expr \rangle)
\end{array}
$$

Judgements $\mathcal{I}$ are of the form

Γ	$\vdash$	*wf*	Γ *is a well formed environment*,
Γ	$\vdash$	A	A *well formed type in* Γ,
Γ	$\vdash$	$E : A$	E *has type* A *in* Γ.

Rules $\mathcal{R}$ are:

$$\frac{}{\emptyset \vdash wf} \qquad [\textsc{Env}\ \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin dom(\Gamma)}{\Gamma, x : A \vdash wf} \qquad [\textsc{Env}\ x]$$

$$\frac{\Gamma \vdash wf \quad A \in base\ type}{\Gamma \vdash A} \qquad [\textsc{Type Const}]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \qquad [\textsc{Type Arrow}]$$

$$\frac{\Gamma', x:A, \Gamma'' \vdash wf}{\Gamma', x:A, \Gamma'' \vdash x:A} \qquad [\textsc{Val } x]$$

$$\frac{\Gamma, x:A \vdash E:B}{\Gamma \vdash \lambda x:A\,.\,E:A \to B} \qquad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E:A \to B \quad \Gamma \vdash F:A}{\Gamma \vdash EF:B} \qquad [\textsc{Val Appl}]$$

The rules of the type system are used for type derivation, for example to deduce the type of the expression $\lambda x : Nat\,.\,x$ we get:

$$\frac{\dfrac{\dfrac{\dfrac{\emptyset \vdash wf \quad Nat \in K}{\emptyset \vdash Nat} \quad x \notin dom(\emptyset)}{x : Nat \vdash wf}}{x : Nat \vdash x : Nat}}{\emptyset \vdash (\lambda x : Nat.x) : Nat \to Nat}$$

As a part of the operational semantics of type system F_1, β-reduction is defined by

$$(\lambda x : A.E)F \to_\beta E[x := F],$$

and the type rule for this reduction is

$$\frac{\Gamma \vdash (\lambda x : A.E)F : B}{\Gamma \vdash E[x := F] : B} \qquad [\textsc{Conv } \beta]$$

The type system F_1 has *Church–Rosser property*, i.e. it is a confluent system. β-reductions preserve the type of expressions:

Theorem 2.1. *If $E : A$ and $E \to_\beta^+ F$, then $F : A$.*

One of the main results is stated by the *theorem of strong normalizing*:

Theorem 2.2. *There is no infinite reduction sequence on any term.*

Three questions for types arise using type system F_1,

1. *Type checking* Given Γ, E and A, is $\Gamma \vdash E : A$ derivable?
2. *Typeability* Given E, find Γ and A such that $\Gamma \vdash E : A$ is derivable.
3. *Inhabitation* Given A, find Γ and E such that $\Gamma \vdash E : A$ is derivable.

2.2. Higher order λ-calculus with subtypes

First we extend the type system F_1 with type variables, this gives the type system F_2. Polymorphic functions are describable in this system using expressions as $E \equiv \Lambda\alpha.F$, and type applications $E[A]$. If $F : B$ then the type of function E is $\forall\alpha.B$.

The theoretical properties of F_2 are essentially the same as of the system F_1, but more restricted mechanism of recursion may be formulated in its calculus. The class of functions definable in F_2 is much larger than the primitive recursive functions, for example, Ackermann's function can be encoded in this system.

Functions in F_2 take a type as argument and return a term. In the next type system F_3 it is possible to write functions from types to types.

In the type system F_3 type abstractions are introduced and type expressions are used for the mapping from types to types. The general form of type expressions are $\Lambda\alpha.B$. If $E \equiv \Lambda\alpha.F$ is a polymorphic function of type $\forall\alpha.B$, then the type of the application $E[A]$ is the type application $(\Lambda\alpha.B)A$.

A basic difference between $\forall\alpha.B$ and $\Lambda\alpha.B$ is that $\forall\alpha.B$ is a type of a polymorphic function, while $\Lambda\alpha.B$ is a function from types to types.

Since type abstractions and applications need to be correct, the type of type variable has to be introduced. This is the type system F_4. The type of types is called *kind*. The constant kind with name $\star$ is the kind of types of terms, and in this type system, if K is a kind, then so is $\star \to K$.

In order to create more general systems, more general kinds must be allowed. By adding quantifications over constructors of successively higher kinds, the systems $F_5, F_6, \ldots$ are obtained. The union of all these systems is F^ω:

$$F^\omega \equiv F_1 \cup F_2 \cup F_3 \cup \ldots$$

where

$$\begin{array}{lcl} \langle \textit{kind} \rangle & ::= & \star \\ & | & \langle \textit{kind} \rangle \ \to \ \langle \textit{kind} \rangle \end{array}$$

and the modified syntax rules are

$$\begin{array}{lcl} \langle \textit{type} \rangle & ::= & \ldots \\ & | & \forall\ \langle \textit{type-variable} \rangle : \ \langle \textit{kind} \rangle \,.\, \langle \textit{type} \rangle \\ & | & \Lambda\ \langle \textit{type-variable} \rangle : \ \langle \textit{kind} \rangle \,.\, \langle \textit{type} \rangle \\ \langle \lambda\textit{-expr} \rangle & ::= & \ldots \\ & | & \Lambda\ \langle \textit{type-variable} \rangle : \ \langle \textit{kind} \rangle \,.\, \langle \lambda\textit{-expr} \rangle \end{array}$$

The typing, kinding and derivation rules are not detailed here.

Introducing the *subtype* relation $A \leq B$, the extension of the type system F^ω with subtyping is called the *type system* $F^\omega_\leq$.

There are a number of further possible extensions of $F^\omega_\leq$, these extensions can be used to model important concepts of object oriented programming. Specially, $F^\omega_\leq$ with records, lists, existential types and recursive types is a basis for constructing formal models of programming.

Introduction of the *depending type* of form

$$\Pi\ \langle \textit{variable} \rangle : \ \langle \textit{type} \rangle \,.\, \langle \lambda\textit{-expr} \rangle$$

leads to the *Calculus of Construction* (and the Martin-Löf type theory), which is the base of theorem proving environments.

3. Intuitionistic logic

The important difference between classical logic and *intuitionistic logic* is that classical logic is concerned with a notion of truth that is absolute, whereas in intuitionistic logic statements are based on the existence of a proof. For example, there is no proof of the law of excluded middle $A \vee \neg A$, no method of proving or disproving for an arbitrary proposition A, therefore the law of excluded middle is not intuitionistically valid.

3.1. Minimal logic

A variant of propositional logic, in which the only logical connective is the implication, is called *minimal logic*. The formulas F are defined by the grammar

$$F ::= V \mid F \to F$$

where V may be any propositional variable.

The constructive nature of this logic may be expressed with the Brouwer–Heyting–Kolmogorov interpretation, which says that

- a variable A is interpreted as an unspecified construction of A,
- a construction of $A \to B$ is a method that transforms a construction of A into a construction of B,

that is, proving a formula is equivalent to creating a construction (proof, function, program) of it.

A construction for $A \to A$ is a function $f \equiv \lambda x : A \,.\, x$, $f\, A \to_\beta A$.

For technical reasons, we use labeled assumptions: if x is a label, A is a formula, then we write $x : A$. Let Γ be a finite set of labeled assumptions. The *judgements* of the intuitionistic minimal logic has the form $\Gamma \vdash_E A$, where A is a formula, E is the construction (proof) for A and Γ is the environment. Thus for the previous example we write

$$\emptyset \vdash_{\lambda x:A \,.\, x} \; A \to A.$$

The *natural deduction* proof system for minimal logic is characterized using the following axiom and three inference rules:

$$\frac{}{\Gamma, x : A \vdash_x A} \qquad [\textsc{Ident Axiom}]$$

$$\frac{\Gamma \vdash_E B}{\Gamma, x : A \vdash_E B} \qquad [\textsc{Assump}\ A]$$

$$\frac{\Gamma, x : A \vdash_E B}{\Gamma \vdash_{\lambda x:A \,.\, E} A \to B} \qquad [\textsc{Impl Intro}]$$

$$\frac{\Gamma \vdash_E A \to B \quad \Gamma \vdash_F A}{\Gamma \vdash_{EF} B} \qquad [\textsc{Impl Elim}]$$

3.2. Curry–Howard isomorphism

It is easy to see that the terms of the type system F_1 are the proofs of the intuitionistic minimal logic, and the types of the expressions are the formulas proved. One can find more connections between these two systems. These correspondences are called the *Curry–Howard isomorphism*.

intuitionistic minimal logic	type system F_1
propositional variable	type variable
$\rightarrow$ logical connective	$\rightarrow$ type constructor
formula (*proposition*)	*type*
assumption	variable
implication introduction	abstraction
implication elimination	application
proof, construction	expression

An interesting consequence of the isomorphism above is that the provability of a formula A is equivalent to the inhabitation of type A. It means that if a formula A has a proof, then there is an expression which has type A. Similarly, proof checking is equivalent to type checking, i.e. if P is a proof of the formula A, then P is a derivation showing that a term has the given type A.

intuitionistic minimal logic	type system F_1
provability	inhabilitation
proof checking	type checking

We state the next theorem without proof.

Theorem 3.1. *There is a closed λ-expression of type A iff A is a provable formula of intuitionistic minimal logic.*

The Curry–Howard isomorphism is summarized by the slogan *formulas as types and proofs as terms*.

It is worth to mention that there is a correspondence between the typed combinatory logic and the Hilbert-style proof system, between the type system extended by dependent types and the first order intuitionistic logic, and also between the type system F_2 and the second-order logic. In general, to each of the systems of Barendregt's cube there belongs an appropriate logic.

4. Program verification

In the previous sections we studied some connections between type systems and logic. Now we turn our attention to the connections between logic and program verification.

The proof that a program satisfies its specification is called *program verification*. Various approaches of program verification have been proposed. One of them is the *operational reasoning*, it is an analysis in terms of execution sequences of the given program. A different approach is called *axiomatic reasoning*: using this method we need a language to specify the program properties. The language is the *language of predicate logic*, consisting of well-formed *formulas*. Axioms and derivation (proof) rules of the logic allow us to prove that the program satisfies the desired properties.

Nowadays *automatic* program verification is subject to intense research. If a program operates only on finite data types, that is, on a finite state space, then automatic program verification is indeed possible. The checking whether the program is a model of its specification is called *model checking*.

4.1. Formal proof system

In the theory of program verification well-formed formulas are used to write the properties of a program, and a *proof system* is needed to show that the program satisfies these properties.

A proof system over a set Φ of formulas is a finite set of axiom schemes and proof rules. If φ is an axiom then it is considered as a given fact, and with the help of rules further facts can be deduced from the formulas. A *proof* of a formula φ in the proof system is a finite sequence

$$\varphi_1 \ \varphi_2 \ \ldots \varphi_n,$$

where φ_1 is an axiom, $\varphi = \varphi_n$, and each formula φ_i $(1 \leq i \leq n)$ is either an axiom or it can be obtained by an application of a rule from formulas of the proof system. In this case φ is a theorem, and we write $\vdash \varphi$.

Formulas from predicate logic are used to write properties of program executions. These formulas are called *assertions*.

Proof theory deals with the correctness formulas, these have the form

$$\{pre\} \ P \ \{post\},$$

where *pre* and *post* are assertions and P is a program. The correctness formula is true in the sense of *total* correctness, if every computation of P that starts in the state satisfying *pre* terminates and its final state satisfies *post*. In the case of *partial correctness*, diverging computations of P are not taken into account, that is, the above sentence is valid for terminating computations only.

4.2. Finding proofs

Theorem provers are based on type theory. The provers use the strategy of goal-directed simplification: to find a proof for a goal, we first find a proof for simpler subgoals, and then we apply an inference rule to proofs of the subgoals that yields to a proof of the original goal.

Such a strategy is encoded formally by a *tactic*, a function that maps a goal to a pair containing a list of subgoals and a validation. A validation maps proofs of the subgoals to a proof of the original goal.

The type of tactics is

$$\mathit{type\ tactic} = \mathit{goal} \to (\mathit{goal\ list} \times (\mathit{proof\ list} \to \mathit{proof})),$$

that is, a tactic returns a list of subgoals and a validation that maps proofs of the subgoals to a proof of the goal. This means that if a tactic is applied to goal g and it returns $([\,], f)$, then $f\,[\,]$ is a proof of g.

The basic tools in proof systems for automatically applying rules are *tactics*. A tactic examines the given goal situation and reduces it to the problem of solving a number of subgoals, and applies rules of inference forwards or backwards to derive new proof formulas or to justify certain formulas of the proof. The main use of tactics is to translate a proof into a natural deduction.

But the provers usually use goal-directed "interactive" methods, in the inference steps the user specifies steps and the details are proved automatically. Using tactics the user can compose so-called theorem proving *primitives* that can be inference rules or calls to decision procedures. For example, PVS has many primitives and a tiny tactic language, while the Isabelle/HOL system has a full programming language (ML) as a tactic language to compose more sophisticated primitives.

If we need to be able to combine primitive tactics via alternation, composition or repetition, we create combinators, which are known as *tacticals*. For example, some fundamental tacticals:

T_1 `then` T_2	*applies* T_1 *and then applies* T_2,
T_1 `orelse` T_2	*tries to apply* T_1 *and, if it fails, applies* T_2,
`repeat` T_1	*repeatedly applies* T_1 *until it fails.*

5. Conclusion

Many widely-used provers are based on some kind of type theory. It was demonstrated that

- types are useful for organizing formal knowledge,
- the type of an object conveys useful information to reasoners,
- the Curry–Howard isomorphism gives a simple method to synthetize programs from proofs.

Unfortunately, formal correctness proofs are rather sophisticated and time consuming. Proving simple things are quite easy, but as soon as theorems get more complex, the complexity of the proofs increases rapidly. Of course training and routine can help to shorten the time required.

One of the advantages of formal proofs is that we get a deep insight into the programs. Sometimes this leads to a better understanding of the algorithms used and to improvements and simplifications of its code and documentation.

References

[1] Apt, K.R., Olderog, E-R.: *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1997.

[2] Csörnyei, Z.: *Type Systems*, Lecture Notes (2003), http://people.inf.elte.hu/csz (In Hungarian)

[3] Dunfield, J., Pfenning, F.: Tridirectional Typechecking, in *POPL'04*, January 14-16, 2004, Venice, Italy

[4] Harper, R., Pfenning, F.: *Type Refinements*, Project Description, 2001. http://www-2.cs.cmu.edu/1triple/triple.pdf

[5] Pierce, B.C.: *Types and Programming Languages*, The MIT Press, 2002.

[6] Schwartzbach, M.I.: Polymorphic Type Inference *BRICS Lecture Series*, LS-95-3 (1995)

[7] Sørensen, M.H.B., Urzyczyn, P.: *Lectures on Curry-Howard Isomorphism*, Lecture Notes, University of Copenhagen, University of Warsaw (1999).

Postal address

Zoltán Csörnyei
Department of Programming Languages
and Compilers
Eötvös Loránd University
Pázmány Péter sétány 1/c
H-1117 Budapest
Hungary