Годишник на секция "Информатика"
Съюз на учените в България
Том 6, 2013, 86–94

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 6, 2013, 86–94

DEPENDENTLY-TYPED PROGRAMMING

**Trifon Trifonov, Atanas Semerdzhiev, Georgi Penchev,
Magdalina Todorova, Maria Nisheva-Pavlova, Petar Armyanov**

Faculty of Mathematics and Informatics, Sofia University
5 James Bourchier blvd., Sofia 1164, Bulgaria

Abstract: The present work presents a survey of the present state of programming languages employing type systems with dependent types, i.e., types parameterized by values. The features of the languages are reviewed and compared with an emphasis on the usefulness of the type checker as a tool for asserting correctness statements about programs.
Keywords: type systems, dependent types, programming languages, program correctness

1. Introduction

In the rapid and dynamic development in the relatively short history of computer science and software engineering, one concept has remained relatively stable: the computational models (such as the Turing machine), which were developed over 75 years ago and describing the limits of what can be automatically computed by a machine. Nevertheless, the task of the nowadays programmer is not easy: there is a constantly changing stream of concepts, paradigms and technologies, which are more suitable and more efficient than their predecessors. Therefore, one of the major tasks of the theoretical computer science is to develop novel programming languages and tools, with which developers can express more efficiently and naturally the logic and the intention behind the created programs.

One of the most important targets for moving the burden from the human to the machine is ensuring of program consistency. Naturally, it is not realistic to hope for a completely automatic tool, which would assert that the program correctly and fully fits the intended purpose of its creator. However, there have already been developed a number of models, in which a certain degree of correctness can be asserted. Such examples are model checking techniques [1] and static program analysis tools [2]. It is often overlooked that one of the simplest and most traditional tools for asserting and verifying program properties are type systems. Arguably, the most important categorization of type systems is the distinction between static type systems, in which type checking and inference is executed during compile-time and dynamic type systems, in which these processes happen in run-time. In the present study we will be most considered in static type systems, as they allow proving properties about the program without executing them, i.e., independently of the program's complexity.

A primary task of type systems is to ensure a property of programs referred to as **type safety**. This is a consistency property, which asserts that a program avoids a certain class of errors, (**type errors**), i.e., there is no execution of a program which leads to a non-terminal state where no valid transition to a next state exists [3]. For example, if a C function has the signature `int f(int x)`, and it is compiled correctly, then the compiler has automatically asserted that every time the function is invoked with an integer, it will return an integer as a result. On the other hand, if the compiler refuses to compile the function because of a type error, this means it has found a counterexample that under certain conditions the property would not hold.

Static type checking is a powerful tool, but it might also impact expressiveness, as it requires programs to follow strict typing rules in order to be accepted by the compiler. One of the radical solutions to increase expressiveness is to transition to a dynamic type system, in which no prior type checking whatsoever is done. Unfortunately, this means giving up the important benefit of asserting a property of the program before executing it. An alternative approach is to use a more expressive type system. A canonical example to this phenomenon is type polymorphism. Consider a C function, which returns the larger of its two arguments:

```
int max(int x, int y) { if (x > y) return x; else return y; }
```

In C we cannot use the same function to find the maximum of two floating point numbers, or of two strings. To this end we would need to write separate functions with the same body but with different signatures. This illustrates a weakness of the C type system which limits the expressiveness of the language. In C++ this problem is (partially) resolved by the introduction of functions with type parameters, i.e., function templates. This allows us to write a function template, which will work correctly for any type for which the operator > is defined:

```
template <typename T>
T max(T x, T y) { if (x > y) return x; else return y; }
```

Despite the fact that C++ templates do not introduce full type polymorphism [4], it is clear that their presence increases the expressiveness of programs and also the complexity of the compiler by providing more automation to assert more complex type properties. Parametric type polymorphism was first introduced in the Standard ML language [5] and was an exotic type system implemented in a relatively small number of programming languages. Nowadays, generics are an essential feature of most mainstream programming languages [6].

This paper focuses on another popular extension of type system: dependent types, or types dependent on values. It is the exact opposite of type polymorphism, in which we have values dependent on type. Although such type systems are relatively well studied [7], they do not enjoy the popularity of generics. However, it is quite possible that dependent type systems will become an important part of commercial programming languages in the future. The goal of the paper is to present the current state of dependently-typed programming languages and to highlight advantages and challenges of programming with dependent types.

2. Dependent type theories

Dependent type systems are often considered in a pure theoretical setting, taking lambda calculus as the underlying computational model. As a result, currently existing dependent type programming languages are functional and not procedural. Two separate sorts of objects are considered: **terms** (programs) and **types**.

To illustrate the underlying idea of dependent types, we will describe it as an extension of the simply-typed lambda calculus ($\lambda\Rightarrow$):

1. every type variable α is a type
2. if ρ and σ are types, then $\rho \Rightarrow \sigma$ is a function type from ρ to σ
3. if x is a variable (from a fixed variable set) and τ is a type, then $x:\tau$ is a typed term
4. if $M:\rho \Rightarrow \sigma$ and $N:\rho$ are typed terms, then $(MN):\sigma$ is a typed term
5. if $M:\sigma$ is a typed term and $x:\rho$ is a typed variable then $\lambda(x:\rho)M:\rho \Rightarrow \sigma$ is a typed term

The system $\lambda\Rightarrow$ provides syntax for constructing objects expressing the most simple kind of dependency: from terms on terms, i.e. lambda functions. In his seminal work [7], Barendregt considers three other possible dependencies:

1. from types to types (type constructors or type functions)
2. from types to terms (polymorphic functions)
3. **from terms to types (dependent types).**

These three extensions are completely orthogonal and can be freely combined, prompting Barendregt to define the lambda cube model, illustrated in Figure 1.

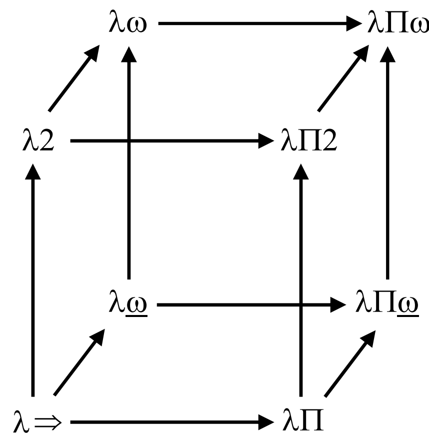

Figure 1. The lambda cube model

The simplest dependent type system is usually denoted as $\lambda\Pi$ and due to its descriptive power is usually being referred to as the Logical Framework (LF). It extends $\lambda\Rightarrow$ by adding a third sort of object — a **kind**. The intended meaning of a kind is to represent "the signature of a type" and to allow types parametrized by values. The base kind * is the signature of a type with no parameters. $\lambda\Pi$ extends the rules of $\lambda\Rightarrow$ as follows:

1. * is a base kind
2. if α is a type variable and k is a kind then α::k is a kinded type
3. if x is a variable and ρ::* is a kinded type, then x:ρ is a typed term
4. if ρ::* is a kinded type and k is a kind, then $\rho \Rightarrow k$ is a kind
5. if x:ρ is a typed variable and σ::* is a kinded type, then $\Pi(x{:}\rho)\sigma$::* is a kinded function type dependent on the parameter x
6. if $\sigma::\rho \Rightarrow k$ is a kinded type and M:ρ is a typed term, then (σM)::k is a kinded type
7. if M:$\Pi(x{:}\rho)\sigma$ and N:ρ are typed terms, then M(N):σ is a typed term
8. if M:σ is a typed term, x:ρ is a typed variable, then $\lambda(x{:}\rho)M{:}\Pi(x{:}\rho)\sigma$ is a typed term

The operator Π is traditionally called a dependent product operator and the type $\Pi(x{:}\rho)\sigma$ is called a dependent product, representing a product of a family of types σ_x, indexed by a variable x of type ρ. As an example, consider a type Vec(n, ρ), which represents a type of n-tuples of elements of type ρ. Then the dependent type Π(n:N)Vec(n, ρ) is the type of a function, which takes a natural number n as an argument and returns an n-tuple of type ρ. Note that this function may return values of different types every time, depending on the argument with which it is called. The dependent product Π can be viewed as a generalization of $\Rightarrow$, since it is the type of a function, which returns a value of a constant type, i.e., not depending on the value of the argument. In other words, $\rho \Rightarrow \sigma$ can be defined as $\Pi(x{:}\rho)\sigma$, where σ does not depend on x.

As shown in the lambda cube (Fig. 1), the basic dependent type system $\lambda\Pi$ can be extended further to the following systems:

1. $\lambda\Pi2$ by adding type polymorphism
2. $\lambda\Pi\underline{\omega}$ by adding type constructors
3. $\lambda\Pi\omega$ by adding type polymorphism and type constructors

The last system $\lambda\Pi\omega$ is also known as Calculus of Constructions [8], which is the basis of the underlying theory of the interactive theorem prover Coq [9].

3. Dependently-typed programming languages and systems

This chapter contains a survey of the currently available programming languages with dependent type systems. The list is not comprehensive, rather it focuses on the more notable implementations of interpreters and compilers of such languages. The presence of dependent types inevitably complicates the corresponding type checkers. However, once the step towards dependent types is already made, it is relatively straightforward to include other extensions of the type systems. Possible extensions include not only the rest of the $\lambda\Pi$-based systems of the lambda cube, but also subtypes [4,10], inductive and coinductive types [9].

The exciting discovery of the natural correspondence between formulas and types, and between proofs and programs, knows as the Curry-Howard correspondence [11] enabled

a fruitful exchange of ideas between logic and theoretical computer science. The logical equivalent of λΠ is the first-order predicate calculus, where dependent types are the natural type equivalent of logical predicates. Therefore, programs in dependent type systems can also be viewed as proofs in first-order predicate logic and its extensions. Similarly, dependent type checkers can be viewed as proof checkers, i.e., programs, which assert the correctness of logical proofs. This is why many of the interpreters and compilers of programming languages with dependent types can also serve as interactive proof assistants. The reason is that the problem of building a program, which is correct with respect to dependent types, or, respectively, a proof of a logical theorem, is interesting and complex enough by itself, even without considering questions of evaluation, interpretation and compilation.

The following sections give a brief overview of several languages, which have dependent-type systems. For each of them a short description is given, together with historical notes, the underlying type system is mentioned, and a summary of the key features, common applications, as well as advantages and disadvantages of the respective language.

3.1. Agda

Agda is a functional programming language with dependent types. It was developed in the University of Chalmers, Sweden. Its development started as early as 1999. Its second version, started in 2007 by Ulf Norell, is a complete rewrite of the system in Haskell [12]. The underlying formal system is Per-Martin Löf's constructive type theory [13]. Agda is a compiled language uses the text editor Emacs as a development environment. Some of the main features of the system are a type checker equipped with partial type inference, natural syntax via unicode support and definition of "mixfix" operators of arbitrary arity, implicit arguments and their automatic inference, pattern matching. Agda is mostly used for modeling various constructions from mathematics and theoretical computer science, for verifying Haskell programs, as well as for building libraries for standard data structures, which are proven correct inside the system. The major advantages of Agda are its clean and rich theoretical foundations, very readable and compact source code, convenient development environment, and the fact that it is still undergoing active development. Among the disadvantages is its relatively young age and limited support for theorem-proving automation, such as proof tactics.

3.2. Coq

Coq is an interactive proof assistant, which has a built-in term language that can be used for dependent-type programming [9]. It was developed in Paris in a joint effort by university and research centers. The development of Coq started in 1984 in an attempt to create an implementation of the more complicated system in the lambda cube, the Calculus of Constructions [8]. Later, the underlying system was extended with inductive types, and the difference between proofs and programs was made explicit by introducing two sorts Prop and Set. Coq is developed in the OCaml functional language and can work in both in interactive mode (as an interpreter) or as a compiler. It comes bundled with its own environment CoqIDE. Among the popular features of Coq are its rich set of proof tactics,

including a high-level language Gallina for developing new tactics, a very small and efficient kernel implementing the core of the system, extraction of dependently-typed programs from proofs, which are correct by construction. Because of its long history, Coq is very well developed, and includes a rich set of standard libraries. Additionally, there are many tools and extensions to Coq, including ones for verifying programs in C and Java. The main application of Coq is creation of formal models, formulating and formally proving properties of them. Although Coq can be used as a dependently-typed programming language, this is not the main focus of the system. Disadvantages of the system include its complexity and generality, which limit the possibility of extensions with new features, if they are not compatible with the core system.

3.3. Epigram

Epigram is a functional programming language with dependent types, which aims to gently introduce the proofs-as-programs concept, while keeping closer to the programming side. Its development started in 2004 by Conor McBride and James McKinna in the university of Nottingham [14]. The underlying theory is Luo's UTT [15], which is essentially constructive type theory with inductive datatypes. A very notable feature of Epigram is its two-dimensional syntax, which allows for a very visual representation of recursive programs as rules. There is also integration with the text editor XEmacs, which allows for easier editing of programs. Epigram is very close to spirit to Agda, but is less popular and with a smaller community. Its main advantage is that the underlying language is very intuitive for people familiar with traditional functional programming, while a disadvantage is its infrastructural immaturity in terms of tools, libraries and support.

3.4. Idris

Idris is a functional programming language with dependent types, similar to Agda and Epigram. It is being actively developed in Haskell by Edwin Brady since 2011 [16]. The underlying system is Per-Martin Löf's type theory [13], and the main focus of the language is to present a framework for building verified systems. It has been designed with interoperability with C in mind and is rich in various syntactic constructions, such as type classes, dependent records, monad comprehensions. Additionally, unlike Agda, it has good support for proof tactics. The main advantages of Idris is its syntactic similarity to Haskell, and its foreign function interface to C. Nevertheless, Idris is quite young and at this point has only one active developer.

3.5. Matita

Matita is an interactive proof assistant, meant to provide an alternative to Coq, as a lightweight implementation of the Calculus of Inductive Constructions [17]. It is essentially a reimplementation of Coq from scratch, again using OCaml as the development language. Matita is being developed at the University of Bologna since 2008. The system has most of the core features of Coq, but in order to reduce complexity some features have been left out, including program extraction, user-defined tactics, inferring of implicit arguments, some tactics and decision procedures. The main advantage of the system is its reduced complexity,

making it easy to extend and improve with less knowledge of the internals. Correspondingly, the major disadvantage is that it is less feature-rich and not widely adopted.

3.6. Twelf

Twelf is a logic programming language employing dependent types [18]. It has been developed since 2006 in the Carnegie-Mellon University. The implementation language is Standard ML. The underlying system is the logical framework λΠ. The main application of Twelf is to formally specify mathematical theories, including metatheories of programming languages, to express properties of those theories and to prove them. Therefore, the focus of the system is strongly shifted to proofs rather than programming. Nevertheless, Twelf can be used for defining dependently typed logical programs, which can be executed by specifying a goal, which performs a search procedure, essentially leading to evaluation of recursively defined programs. The main advantage of the system is the simplicity underlying theory, which is at the same time is very expressive. However, Twelf is not suited for general-purpose programming, as it lacks some basic conveniences for efficiency (such as cut) and built-in functions (even for input and output).

4. Conclusion

Dependent-type programming is an immersing field in computer science, enjoying extensive attention from both theoretical computer scientists and logicians. The level of complexity allows for a wider variety of approaches to implementing dependent-type systems, which is also visible from the languages reviewed in the present paper. Despite the active ongoing development on many fronts and the continuously growing community, at this point practical programming with dependent types remains from mainstream commercial use.

From our analysis, the following reasons for this state can be deduced:

1. a high level of complexity of the theory, requiring formal training and extensive expertise
2. diversified field with many competing approaches
3. relatively young discipline with only a small number of examples for successful practical applications

Despite the challenges described above, we have also identified many advantages of dependent-type programming, which have the potential to make the field more popular:

1. automated correctness verification by type checking
2. automated and interactive program construction by means of type inference and implicit arguments
3. close relation to mathematical logic, allowing reuse of results and approaches
4. high level of expressiveness, allowing for modeling complicated structures and concepts with shorter programs.

As next steps, we will examine and compare the practical aspects of the languages described above in more detail, in the light of a set of case studies, which highlight the advantages of dependent-type programming.

Acknowledgements

The presented study has been supported by the Bulgarian National Science Fund within the project titled "Contemporary programming languages, environments and technologies and their application in building up software developers", Grant No. DFNI-I01/12.

References

[1] Clarke, E., O. Grumberg, D. Long. Model checking and abstraction, *ACM Transactions on Programming Languages and Systems* (TOPLAS) Vol. 16, 1994, No. 5, 1512–1542.

[2] Hankin, C., F. Nielson, H. Nielson. *Principles of Program Analysis*, Springer, 1999.

[3] Pierce, B. *Types and programming languages*, The MIT Press, 2002.

[4] Stroustrup, B. C++ *Programming Language, 3th edition*, Pearson Education India, 1994.

[5] Milner, R. A theory of type polymorphism in programming, J*ournal of computer and system sciences*, Vol. 17, No. 3, 1978, 348–375.

[6] Reynolds, J. *Theories of programming languages*, Cambridge University Press, Chapter 16, 1998.

[7] Barendregt, H., W. Dekkers, R. Statman. Lambda calculus with types. In: *Handbook of logic in computer science, Vol. 2*, Oxford University Press, USA, 1992, 118–310.

[8] Coquand, T., G. Huet. *The calculus of constructions*, INRIA Research Report, 1986.

[9] Bertot, Y., P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, Springer-Verlag New York Incorporated, 2004.

[10] M. Abadi, L. Cardelli, *A theory of objects*, Springer, 1996.

[11] Howard, W. The formulae-as-types notion of construction, In: Seldin, Jonathan P.; Hindley, J. Roger, To H.B. Curry: *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Boston, MA: Academic Press, 1980, 479–490.

[12] Norell, U. Dependently typed programming in Agda. *Advanced Functional Programming*, 2009, 230–266.

[13] Martin-Lof, P. *Intuitionistic type theory*. Notes by Giovanni Sambin. Naples,, Italy: Bibliopolis, 1984.

[14] McBride, C.. Epigram: Practical programming with dependent types. *Advanced Functional Programming*, 2005, 130–170.

[15] Luo, Z. *Computation and reasoning: a type theory for computer science*, Oxford University Press, Inc., 1994.

[16] Brady, E. IDRIS: systems programming meets full dependent types. In: *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, 2011, 43–54.

[17] Asperti, A., W. Ricciotti, C. S. Coen, E. Tassi. The Matita interactive theorem prover. *Automated Deduction–CADE*-23, 2011, 64–69.

[18] Pfenning, F., C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. *Automated Deduction—CADE*-16, 2009, 202–206.