WIKIPEDIA
The Free Encyclopedia

Dependent type

In computer science and logic, a **dependent type** is a type whose definition depends on a value. It is an overlapping feature of type theory and type systems. In intuitionistic type theory, dependent types are used to encode logic's quantifiers like "for all" and "there exists". In functional programming languages like Agda, ATS, Coq, F*, Epigram, Idris, and Lean, dependent types help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.

Two common examples of dependent types are *dependent functions* and *dependent pairs*. The return type of a dependent function may depend on the *value* (not just type) of one of its arguments. For instance, a function that takes a positive integer n may return an array of length n, where the array length is part of the type of the array. (Note that this is different from polymorphism and generic programming, both of which include the type as an argument.) A dependent pair may have a second value the type of which depends on the first value. Sticking with the array example, a dependent pair may be used to pair an array with its length in a type-safe way.

Dependent types add complexity to a type system. Deciding the equality of dependent types in a program may require computations. If arbitrary values are allowed in dependent types, then deciding type equality may involve deciding whether two arbitrary programs produce the same result; hence the decidability of type checking may depend on the given type theory's semantics of equality, that is, whether the type theory is intensional or extensional.[1]

History

In 1934, Haskell Curry noticed that the types used in typed lambda calculus, and in its combinatory logic counterpart, followed the same pattern as axioms in propositional logic. Going further, for every proof in the logic, there was a matching function (term) in the programming language. One of Curry's examples was the correspondence between simply typed lambda calculus and intuitionistic logic.[2]

Predicate logic is an extension of propositional logic, adding quantifiers. Howard and de Bruijn extended lambda calculus to match this more powerful logic by creating types for dependent functions, which correspond to "for all", and dependent pairs, which correspond to "there exists".[3]

(Because of this and other work by Howard, propositions-as-types is known as the Curry–Howard correspondence.)

Formal definition

Loosely speaking, dependent types are similar to the type of an indexed family of sets. More formally, given a type $A : \mathcal{U}$ in a universe of types $\mathcal{U}$, one may have a **family of types** $B : A \to \mathcal{U}$, which assigns to each term $a : A$ a type $B(a) : \mathcal{U}$. We say that the type $B(a)$ varies with a.

Π type

A function whose type of return value varies with its argument (i.e. there is no fixed codomain) is a **dependent function** and the type of this function is called **dependent product type**, **pi-type** (**Π type**) or **dependent function type**.[4] From a family of types $B : A \to \mathcal{U}$ we may construct the type of dependent functions $\prod_{x:A} B(x)$, whose terms are functions that take a term $a : A$ and return a term in $B(a)$. For this example, the dependent function type is typically written as $\prod_{x:A} B(x)$ or $\prod (x : A) B(x)$.

If $B : A \to \mathcal{U}$ is a constant function, the corresponding dependent product type is equivalent to an ordinary function type. That is, $\prod_{x:A} B$ is judgmentally equal to $A \to B$ when B does not depend on x.

The name 'Π-type' comes from the idea that these may be viewed as a Cartesian product of types. Π-types can also be understood as models of universal quantifiers.

For example, if we write $\mathrm{Vec}(\mathbb{R}, n)$ for n-tuples of real numbers, then $\prod_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ would be the type of a function which, given a natural number n, returns a tuple of real numbers of size n. The usual function space arises as a special case when the range type does not actually depend on the input. E.g. $\prod_{n:\mathbb{N}} \mathbb{R}$ is the type of functions from natural numbers to the real numbers, which is written as $\mathbb{N} \to \mathbb{R}$ in typed lambda calculus.

For a more concrete example, taking A to be the type of unsigned integers from 0 to 255 (the ones that fit into 8 bits or 1 byte) and $B(a) = X_a$ for $a : A$, then $\prod_{x:A} B(x)$ devolves into the product of $X_0 \times X_1 \times X_2 \times ... \times X_{253} \times X_{254} \times X_{255}$.

Σ type

The dual of the dependent product type is the **dependent pair type**, **dependent sum type**, **sigma-type**, or (confusingly) **dependent product type**.[4] Sigma-types can also be understood as existential quantifiers. Continuing the above example, if, in the universe of types $\mathcal{U}$, there is a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$, then there is a dependent pair type $\sum_{x:A} B(x)$. (The alternative notations are similar to that of Π types.)

The dependent pair type captures the idea of an ordered pair where the type of the second term is dependent on the value of the first. If $(a, b) : \sum_{x:A} B(x)$, then $a : A$ and $b : B(a)$. If B is a constant function, then the dependent pair type becomes (is judgementally equal to) the product type, that is, an ordinary Cartesian product $A \times B$.[4]

For a more concrete example, taking A to again be type of unsigned integers from 0 to 255, and $B(a)$ to again be equal to X_a for 256 more arbitrary X_a, then $\sum_{x:A} B(x)$ devolves into the sum $X_0 + X_1 + X_2 + ... + X_{253} + X_{254} + X_{255}$.

Example as existential quantification

Let $A : \mathcal{U}$ be some type, and let $B : A \to \mathcal{U}$. By the Curry–Howard correspondence, B can be interpreted as a logical predicate on terms of A. For a given $a : A$, whether the type $B(a)$ is inhabited indicates whether a satisfies this predicate. The correspondence can be extended to existential quantification and dependent pairs: the proposition $\exists a \in A\, B(a)$ is true if and only if the type $\sum_{a:A} B(a)$ is inhabited.

For example, $m : \mathbb{N}$ is less than or equal to $n : \mathbb{N}$ if and only if there exists another natural number $k : \mathbb{N}$ such that $m + k = n$. In logic, this statement is codified by existential quantification:

$$m \leq n \iff \exists k{:}\mathbb{N}\, m + k = n.$$

This proposition corresponds to the dependent pair type:

$$\sum_{k:\mathbb{N}} m + k = n.$$

That is, a proof of the statement that m is less than or equal to n is a pair that contains both a non-negative number k, which is the difference between m and n, and a proof of the equality $m + k = n$.

Systems of the lambda cube

Henk Barendregt developed the lambda cube as a means of classifying type systems along three axes. The eight corners of the resulting cube-shaped diagram each correspond to a type system, with simply typed lambda calculus in the least expressive corner, and calculus of constructions in the most expressive. The three axes of the cube correspond to three different augmentations of the simply typed lambda calculus: the addition of dependent types, the addition of polymorphism, and the addition of higher kinded type constructors (functions from types to types, for example). The lambda cube is generalized further by pure type systems.

First order dependent type theory

The system $\lambda\Pi$ of pure first order dependent types, corresponding to the logical framework LF, is obtained by generalising the function space type of the simply typed lambda calculus to the dependent product type.

Second order dependent type theory

The system $\lambda\Pi 2$ of second order dependent types is obtained from $\lambda\Pi$ by allowing quantification over type constructors. In this theory the dependent product operator subsumes both the $\rightarrow$ operator of simply typed lambda calculus and the $\forall$ binder of System F.

Higher order dependently typed polymorphic lambda calculus

The higher order system $\lambda\Pi\omega$ extends $\lambda\Pi 2$ to all four forms of abstraction from the lambda cube: functions from terms to terms, types to types, terms to types and types to terms. The system corresponds to the calculus of constructions whose derivative, the calculus of inductive constructions is the underlying system of the Coq proof assistant.

Simultaneous programming language and logic

The Curry–Howard correspondence implies that types can be constructed that express arbitrarily complex mathematical properties. If the user can supply a constructive proof that a type is *inhabited* (i.e., that a value of that type exists) then a compiler can check the proof and convert it into executable computer code that computes the value by carrying out the construction. The proof checking feature makes dependently typed languages closely related to proof assistants. The code-generation aspect provides a powerful approach to formal program verification and proof-carrying code, since the code is derived directly from a mechanically verified mathematical proof.

Comparison of languages with dependent types

Language	Actively developed	Paradigm[a]	Tactics	Proof terms	Termination checking	Types can depend on[b]	Universes	Proof irrelevance	Program extraction	Extraction erases irrelevant terms
Ada 2012 (http://www.ada-auth.org/standards/ada12.html)	Yes[5]	Imperative	Yes[6]	No	?	Any term[c]	?	?	Ada	?
Agda	Yes[7]	Purely functional	Few/ limited[d]	Yes	Yes (optional)	Any term	Yes (optional)[e]	Proof-irrelevant arguments[9] Proof-irrelevant propositions[10]	Haskell, JavaScript	Yes[9]
ATS	Yes[11]	Functional / imperative	No[12]	Yes	Yes	Static terms[13]	?	Yes	Yes	Yes
Cayenne	No	Purely functional	No	Yes	No	Any term	No	No	?	?
Gallina (Coq)	Yes[14]	Purely functional	Yes	Yes	Yes	Any term	Yes[f]	Yes[15]	Haskell, Scheme and OCaml	Yes
Dependent ML	No[g]	?	?	Yes	?	Natural numbers	?	?	?	?
F*	Yes[16]	Functional and imperative	Yes[17]	Yes	Yes (optional)	Any pure term	Yes	Yes	OCaml, F#, and C	Yes
Guru (https://code.google.com/p/guru-lang/)	No[18]	Purely functional[19]	hypjoin[20]	Yes[19]	Yes	Any term	No	Yes	Carraway	Yes
Idris	Yes[21]	Purely functional[22]	Yes[23]	Yes	Yes (optional)	Any term	Yes	No	Yes	Yes[23]
Lean	Yes	Purely functional	Yes	Yes	Yes	Any term	Yes	Yes	Yes	Yes
Matita	Yes[24]	Purely functional	Yes	Yes	Yes	Any term	Yes	Yes	OCaml	Yes
NuPRL	Yes	Purely functional	Yes	Yes	Yes	Any term	Yes	?	Yes	?
PVS	Yes	?	Yes	?	?	?	?	?	?	?
Sage (http://sage.soe.ucsc.edu/) Archived (https://web.archive.org/web/20201109032232/http://sage.soe.ucsc.edu/) 2020-11-09 at the Wayback Machine	No[h]	Purely functional	No	No	No	?	No	?	?	?
Twelf	Yes	Logic programming	?	Yes	Yes (optional)	Any (LF) term	No	No	?	?

a. This refers to the *core* language, not to any tactic (theorem proving procedure) or code generation sublanguage.
b. Subject to semantic constraints, such as universe constraints
c. Static_Predicate for restricted terms, Dynamic_Predicate for Assert-like checking of any term in type cast
d. Ring solver[8]
e. Optional universes, optional universe polymorphism, and optional explicitly specified universes
f. Universes, automatically inferred universe constraints (not the same as Agda's universe polymorphism) and optional explicit printing of universe constraints
g. Has been superseded by ATS
h. Last Sage paper and last code snapshot are both dated 2006

See also

- Typed lambda calculus
- Intuitionistic type theory

References

1. Hofmann, Martin (1995), *Extensional concepts in intensional type theory* (https://ncatlab.org/nlab/files/HofmannExtensionalIntensionalTypeTheory.pdf) (PDF)
2. Sørensen, Morten Heine B.; Urzyczyn, Pawel (1998), *Lectures on the Curry-Howard Isomorphism*, CiteSeerX 10.1.1.17.7385 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385)
3. Bove, Ana; Dybjer, Peter (2008), *Dependent Types at Work* (http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf) (PDF), Chalmers University of Technology
4. Altenkirch, Thorsten; Danielsson, Nils Anders; Löh, Andres; Oury, Nicolas (2010). "ΠΣ: Dependent Types without the Sugar" (https://www.cs.nott.ac.uk/~psztxa/publ/pisigma-new.pdf) (PDF). In Blume, Matthias; Kobayashi, Naoki; Vidal, Germán (eds.). *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. Lecture Notes in Computer Science. Vol. 6009. Springer. pp. 40–55. doi:10.1007/978-3-642-12251-4_5 (https://doi.org/10.1007%2F978-3-642-12251-4_5).
5. "GNAT Community download page" (https://www.adacore.com/download/).
6. "§3.2.4 Subtype Predicates" (http://www.ada-auth.org/standards/12rm/html/RM-3-2-4.html). *Ada Reference Manual* (2012 ed.).
7. "Agda download page" (http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Download).
8. "Agda Ring Solver" (http://www.cs.nott.ac.uk/~nad/listings/lib/Algebra.RingSolver.html).
9. "Announce: Agda 2.2.8" (https://web.archive.org/web/20110718034319/http://permalink.gmane.org/gmane.comp.lang.agda/2051). Archived from the original (http://permalink.gmane.org/gmane.comp.lang.agda/2051) on 2011-07-18. Retrieved 2010-09-28.
10. "Agda 2.6.0 changelog" (http://hackage.haskell.org/package/Agda-2.6.0/changelog).
11. "ATS2 downloads" (http://sourceforge.net/projects/ats2-lang/files/).
12. "email from ATS inventor Hongwei Xi" (http://sourceforge.net/mailarchive/message.php?msg_id=27050673).
13. Xi, Hongwei (March 2017). "Applied Type System: An Approach to Practical Programming with Theorem-Proving" (http://www.ats-lang.org/MYDATA/ATSfoundation.pdf) (PDF). arXiv:1703.08683 (https://arxiv.org/abs/1703.08683).
14. "Coq CHANGES in Subversion repository" (https://gforge.inria.fr/scm/viewvc.php/trunk/CHANGES?root=coq&view=log).
15. "Introduction of SProp in Coq 8.10" (https://coq.github.io/doc/master/refman/changes.html#version-8-10).
16. "F* changes on GitHub" (https://github.com/FStarLang/FStar/commits/master). *GitHub*.
17. "F* v0.9.5.0 release notes on GitHub" (https://github.com/FStarLang/FStar/releases/tag/v0.9.5.0). *GitHub*.
18. "Guru SVN" (https://code.google.com/p/guru-lang/source/list).
19. Aaron Stump (6 April 2009). "Verified Programming in Guru" (https://web.archive.org/web/20091229234011/http://guru-lang.googlecode.com/svn/branches/1.0/doc/book.pdf) (PDF). Archived from the original (http://guru-lang.googlecode.com/svn/branches/1.0/doc/book.pdf) (PDF) on 29 December 2009. Retrieved 28 September 2010.
20. Petcher, Adam (May 2008). *Deciding Joinability Modulo Ground Equations in Operational Type Theory* (http://www.cs.uiowa.edu/~astump/papers/petcher-thesis.pdf) (PDF) (MSc). Washington University. Retrieved 14 October 2010.
21. "Idris git repository" (https://github.com/idris-lang/Idris-dev/). *GitHub*. 17 May 2022.
22. Brady, Edwin. "Idris, a language with dependent types — extended abstract" (https://eb.host.cs.st-andrews.ac.uk/drafts/ifl08.pdf) (PDF). CiteSeerX 10.1.1.150.9442 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.150.9442).
23. Brady, Edwin. "How does Idris compare to other dependently-typed programming languages?" (https://www.quora.com/How-does-Idris-compare-to-other-dependently-typed-programming-languages).
24. "Matita SVN" (https://web.archive.org/web/20060508150758/http://helm.cs.unibo.it/websvn/listing.php?repname=helm&path=%2F&sc=0). Archived from the original (http://helm.cs.unibo.it/websvn/listing.php?repname=helm&path=%2F&sc=0) on 2006-05-08. Retrieved 2010-09-29.

Further reading

- Martin-Löf, Per (1984). *Intuitionistic Type Theory* (https://www.cs.cmu.edu/afs/cs/Web/People/crary/819-f09/Martin-Lof80.pdf) (PDF). Bibliopolis.
- Nordström, Bengt; Petersson, Kent; Smith, Jan M. (1990). *Programming in Martin-Löf's Type Theory: An Introduction* (http://www.cse.chalmers.se/research/group/logic/book/). Oxford University Press. ISBN 9780198538141.
- Barendregt, H. (1992). "Lambda calculi with types" (ftp://ftp.cs.ru.nl/pub/CompMath.Found/HBK.ps). In Abramsky, S.; Gabbay, D.; Maibaum, T. (eds.). *Handbook of Logic in Computer Science*. Oxford Science Publications. doi:10.1017/CBO9781139032636 (https://doi.org/10.1017%2FCBO9781139032636). hdl:2066/17231 (https://hdl.handle.net/2066%2F17231).
- Brandl, Helmut (2022). Calculus of Constructions (https://hbr.github.io/Lambda-Calculus/cc-tex)
- McBride, Conor; McKinna, James (January 2004). "The view from the left" (http://strictlypositive.org/view.ps.gz). *Journal of Functional Programming*. **14** (1): 69–111. doi:10.1017/s0956796803004829 (https://doi.org/10.1017%2Fs0956796803004829). S2CID 6232997 (https://api.semanticscholar.org/CorpusID:6232997).
- Altenkirch, Thorsten; McBride, Conor; McKinna, James (2006). "Why dependent types matter" (http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf) (PDF). *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13*. ISBN 1-59593-027-2.
- Norell, Ulf (September 2007). *Towards a practical programming language based on dependent type theory* (http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf) (PDF) (PhD). Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology. ISBN 978-91-7291-996-9.
- Oury, Nicolas; Swierstra, Wouter (2008). "The Power of Pi" (https://www.cs.ru.nl/~wouters/Publications/ThePowerOfPi.pdf) (PDF). *ICFP '08: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. pp. 39–50. doi:10.1145/1411204.1411213 (https://doi.org/10.1145%2F1411204.1411213). ISBN 9781595939197. S2CID 16176901 (https://api.semanticscholar.org/CorpusID:16176901).
- Norell, Ulf (2009). "Dependently Typed Programming in Agda" (http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf) (PDF). In Koopman, P.; Plasmeijer, R.; Swierstra, D. (eds.). *Advanced Functional Programming. AFP 2008*. Lecture Notes in Computer Science. Vol. 5832. Springer. pp. 230–266. doi:10.1007/978-3-642-04652-0_5 (https://doi.org/10.1007%2F978-3-642-04652-0_5). ISBN 978-3-642-04651-3.
- Sitnikovski, Boro (2018). *Gentle Introduction to Dependent Types with Idris*. Lean Publishing. ISBN 978-1723139413.
- McBride, Conor; Nordvall-Forsberg, Fredrik (2022). "Type systems for programs respecting dimensions" (https://strathprints.strath.ac.uk/76626/1/McBride_etal_amctmtxii2021_type_systems_for_programs_respecting_dimensions.pdf) (PDF). *Advanced Mathematical and Computational Tools in Metrology and Testing XII*. Advances in Mathematics for Applied Sciences. World Scientific. pp. 331–345. doi:10.1142/9789811242380_0020 (https://doi.org/10.1142%2F9789811242380_0020). ISBN 9789811242380. S2CID 243831207 (https://api.semanticscholar.org/CorpusID:243831207).

External links

- Dependently Typed Programming 2008 (https://web.archive.org/web/20120831123017/http://sneezy.cs.nott.ac.uk/darcs/DTP08/)

- Dependently Typed Programming 2010 (https://web.archive.org/web/20120901113229/http://sneezy.cs.nott.ac.uk/darcs/dtp10/)
- Dependently Typed Programming 2011 (https://www.cs.ru.nl/dtp11/)
- "Dependent type" (http://www.haskell.org/haskellwiki/Dependent_type) at the Haskell Wiki
- dependent type theory (https://ncatlab.org/nlab/show/dependent+type+theory) at the *n*Lab
- dependent type (https://ncatlab.org/nlab/show/dependent+type) at the *n*Lab
- dependent product type (https://ncatlab.org/nlab/show/dependent+product+type) at the *n*Lab
- dependent sum type (https://ncatlab.org/nlab/show/dependent+sum+type) at the *n*Lab
- dependent product (https://ncatlab.org/nlab/show/dependent+product) at the *n*Lab
- dependent sum (https://ncatlab.org/nlab/show/dependent+sum) at the *n*Lab

---