Dependent Types

A quest in proving software properties at compile time

Lapo Toloni

Seminar for the course Software Verification and Validation.
MsC in Computer Science, Università di Pisa, a.a. 2018/2019

29th, April 2019

Mathematical proofs for formal verification of software

A formal verification, or certification, of a computer program is a formalized guarantee – **a proof!** – that the program satisfies given specified properties.

Certification can take the form of a proof that a program has a specified type when interpreted as a term of some sort

Model checking and Type Systems are the two prevalent approaches to program verification.

Model Checking vs Type systems - Definitions

Definition (Model Checking)

Model checking is a **formal** verification technique which allows for properties of a given system to be verified **through a systematic inspection of all the states of the model** used to represent the system.

Definition (Type System)

Type Systems are a **set of rules** that assigns a property called type to the various constructs of a computer program.
Types formalize and enforce implicit categories which programmers use for data structures and other program components.

Model Checking vs Type systems - Definitions

Definition (Model Checking)

Model checking is a **formal** verification technique which allows for properties of a given system to be verified **through a systematic inspection of all the states of the model** used to represent the system.

Definition (Type System)

Type Systems are a **set of rules** that assigns a property called type to the various constructs of a computer program.
Types formalize and enforce implicit categories which programmers use for data structures and other program components.

Model Checking vs Type systems - Comparison

- model checking is performed in a **semantic and whole-program style**
- model checkers are **flow sensitive**.
- model checkers are good at explaining **why a program is rejected**.

- type systems are defined in a **syntactic and modular style**
- type systems are **flow-insensitive**.
- type systems are good at explaining **why a program is accepted**.

The need of expressive languages

Languages based on **highly expressive type theories** are a natural framework to do certified programming “natively”.

Programming in such rich languages is equivalent to build proofs in certain corresponding logical frameworks.

Using this means programs and their certification proof are written at the same time.
We do not have to use any other external piece of code.

A gap to bridge: How to relate type systems to formal verification?

How can we make type systems a tool able to produce certified software?

Curry-Howad Isomorphism

Curry-Howard isomorphism justifies the usage of type theories as a formal tool in the practical context of formal programming.

The Curry-Howard isomorphism states **the intimate correspondence** between

- λ-**calculus**, the model of computation exploited in type theory
- **systems of formal logic** encountered in proof theory

The correspondence at the syntactic level ensures these equivalences:

- formula/propositions $\Leftrightarrow$ types
- proofs $\Leftrightarrow$ terms/programs
- proof normalization $\Leftrightarrow$ term reduction
- provability $\Leftrightarrow$ inhabitation

Curry-Howad Isomorphism

Curry-Howard isomorphism justifies the usage of type theories as a formal tool in the practical context of formal programming.

The Curry-Howard isomorphism states **the intimate correspondence** between

- λ-**calculus**, the model of computation exploited in type theory
- **systems of formal logic** encountered in proof theory

The correspondence at the syntactic level ensures these equivalences:

- formula/propositions $\Leftrightarrow$ types
- proofs $\Leftrightarrow$ terms/programs
- proof normalization $\Leftrightarrow$ term reduction
- provability $\Leftrightarrow$ inhabitation

We accept the "Propositions-as-Types" assumption

Propositions-as-Types

- A constructive proof of a formula is itself a program
- Propositions are Types
- Proofs are Terms

This means that the standard in literature **type judgement**:

$$\Gamma, M : A$$

can be interpreted in the two following ways:

- in the context Γ, A is a type and M is a program of type A
- in the model Γ, A is a proposition and M is a proof of A

The correspondence at the level of formulas & types

Logic side	**Programming side**
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type

Simply typed λ-calculus fully corresponds to (First Order) Propositional Calculus.
(where conjunction and disjunction may be derived from implication)

Simply typed λ-calculus is not enough

The logic system we got using simply typed λ-calculus is not enough.
We need **more refined type theories** to get more expressive logics via Curry Howard.

Increasing Expressiveness of type systems

- Simply typed λ calculus ($\lambda \rightarrow$)
  where **terms may depend on terms**.
  The only abstraction we have is **function abstraction**
- Polymorphism (λP)
  where **terms may depend on types**.
  The newly introduced abstraction are **parametric polymorphic types**.
- Type Operators ($\lambda 2$)
  where **types may depend on types**.
  The newly introduced abstraction are **type constructors**.

Increasing Expressiveness of type systems

- Simply typed λ calculus ($\lambda \rightarrow$)
  where **terms may depend on terms**.
  The only abstraction we have is **function abstraction**
- Polymorphism (λP)
  where **terms may depend on types**.
  The newly introduced abstraction are **parametric polymorphic types**.
- Type Operators ($\lambda 2$)
  where **types may depend on types**.
  The newly introduced abstraction are **type constructors**.

Increasing Expressiveness of type systems

- Simply typed λ calculus ($\lambda \rightarrow$)
  where **terms may depend on terms**.
  The only abstraction we have is **function abstraction**
- Polymorphism (λP)
  where **terms may depend on types**.
  The newly introduced abstraction are **parametric polymorphic types**.
- Type Operators ($\lambda 2$)
  where **types may depend on types**.
  The newly introduced abstraction are **type constructors**.

Dependent Types

Definition (Dependent Types)

Dependent types are types that depend on values.

From this simple definition follows that by using dependent types **we will be able to encode arbitrary mathematical propositions**.

Some basic classic examples of types that can be expressed are:

- fixed length lists.
- set $A \subseteq \mathbb{N}$ of all the $a \leq k$, $a, k \in \mathbb{N}$.

⋆ **programs which implement a particular specification.**

Dependent Types

Definition (Dependent Types)

Dependent types are types that depend on values.

From this simple definition follows that by using dependent types **we will be able to encode arbitrary mathematical propositions**.

Some basic classic examples of types that can be expressed are:

- fixed length lists.
- set $A \subseteq \mathbb{N}$ of all the $a \leq k$, $a, k \in \mathbb{N}$.
- ⋆ **programs which implement a particular specification.**

The correspondence at the level of formulas & types ($\lambda\Pi$)

Logic side	**Programming side**
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type
universal quantification	dependent product type (Π type)
existential quantification	dependent sum type (Σ type)

Dependent types lift the isomorphism up to First-order predicate logic.

Type theories: programming languages and logic systems

The Curry–Howard correspondence implies that types can be constructed so as to express arbitrarily complex mathematical properties.

A type checker of a dependent type system **can check for software properties statically at compile time**.

The code-generation feature provided by these languages gives a powerful approach to formal program verification and **proof-carrying code**, since the code is derived directly from a mechanically verified mathematical proof.

Dependent Types Theory MVPs: Π-type & Σ-type

Dependent types are families of types which vary over the elements (terms) of some other type.

The two principal instances of dependent types are **dependent functions** (Π-types) and **dependent pairs** (Σ-type).

The terms of a Π-type are type functions whose codomain type can vary depending on the element of the domain to which the function is applied

Σ-types are a generalization of product types in which the type of the second component of the pair is allowed to vary dependending on the value of the first component.

Dependent Function (Π-type) - Notation

Dependent function types are similar to the type of an indexed family of sets.

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 type $B(a)$ depends on the term a.

Dependent function types are written as: $\prod_{x:A} B(x)$

If $B : A \to \mathcal{U}$ is a constant function, then the dependent function type is equivalent to an ordinary function type.

It follows that Π-type is the generalization of function types.

Dependent Pair (Σ-type) - Notation

The dual of the dependent function type is the dependent pair type. In set theory it corresponds to an indexed sum (disjoint union) of types.

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 dependent pair type captures the idea of a 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 is the ordinary (Cartesian) product type $A \times B$

Π-type and Σ-type examples

Π) let $\boldsymbol{Vect}(\mathbb{R}, \boldsymbol{k})$ be the type of k-tuples of real numbers, then

$$\prod_{n:\mathbb{N}} Vect(\mathbb{R}, n)$$

would be the type of a dependent function which, given a natural number n, returns the type of tuples of reals of size n.

Σ) let $\boldsymbol{OrdPair}(\boldsymbol{a}, \boldsymbol{b})$ be the type of pairs where the first element is less than the second element, then

$$\sum_{(a,b):\mathbb{N}\times\mathbb{N}} \mathbf{Less}(a, b)$$

would be the type of a dependent pair which couples two natural numbers a and b with a proof of $\boldsymbol{a < b}$

Π-type and Σ-type can be interpreted in two ways

By **propositions-as-types** assumption

- Π-types can be interpreted as universal quantifiers ∀.
- Σ-types can be interpreted as existential quantifiers ∃.

So every dependent type can be read either as a proposition or as a set. This gives us two readings:

1. a dependent type $P : \mathbb{N} \to \mathcal{U}$ is like a family of sets $P(0), P(1), P(2), \ldots$
2. a dependent type $P : \mathbb{N} \to \mathcal{U}$ is a property of natural numbers where we think of $P(n)$ as the proofs that n has property P

[SECTION 3]

Idris

Idris - Haskell with dependent types?

Idris is a **general purpose functional language**

- It means it is basically similar to standard most used functional languages (Haskell, ML-Family)
- It has an almost equal syntax to Haskell, the entry point of a program is called main, IO is managed with a monad and there is do notation.
- Unlike Haskell the evaluation is aeger and all top-level functions must have a type signature.
  (This is due to type inference being, in general, undecidable for languages with dependent types.)

Idris is built on a tiny codebase

The kernel of the IDRIS implementation consists of

- a type checker (with a totality checker embedded in it)
- an evaluator for the intermediate language TT
- a pattern match compiler

This core is implemented in less than 1000 lines of Haskell code. It is important that this kernel remains small because the correctness of the language implementation relies on the correctness of the underlying type system.

One syntax for two sorts

Idris provides full dependent types native support.

The core of the idea is that types are first order citizens of the language, **types can be computed from any expression**.

The language of the run-time system and the language of the type system are one and the same.
There is no difference in what kind of expressions we are allowed to write in function definitions and in function signatures.

A simple example of type computations

We can use types in expressions and therefore write functions that calculate types.

```
StringOrInt : Bool -> Type -- simple example of
StringOrInt False = String -- type level function
StringOrInt True = Int
```

Using this, we can write a function where **the return type is calculated** depending on an input value.

```
getTheAnswerToEverything : (isInt : Bool)
                          -> StringOrInt isInt
getTheAnswerToEverything False = "42"
getTheAnswerToEverything True  =  42
```

A simple example of type computations

We can use types in expressions and therefore write functions that calculate types.

```
StringOrInt : Bool -> Type -- simple example of
StringOrInt False = String -- type level function
StringOrInt True = Int
```

Using this, we can write a function where **the return type is calculated** depending on an input value.

```
getTheAnswerToEverything : (isInt : Bool)
                         -> StringOrInt isInt
getTheAnswerToEverything False = "42"
getTheAnswerToEverything True  =  42
```

Dependent types checker needs total functions

Idris distinguishes between total and partial functions.

A total function is a function that either:

- Terminates for all possible inputs.
- Produces a non-empty, finite, prefix of a possibly infinite result.

Totality is required to have a terminating type checking. **(*)**

And a **terminating type checking implies that**, due to the Curry Howard isomorphism, **we can consider types as proofs** that our functions are correct.

Idris type checker and Totality

It is best to consider type-level functions in exactly the same way as ordinary functions.

There are a couple of technical differences that are useful to know about:

- Type-level functions exist at compile time only.
- Only total functions will be evaluated at type level.

Idris, to ensure that type-checking itself terminates even in real world cases, treats functions that are not total as constants at the type level, leaving the programmer with some unverified code.

The standard hello world of dependent types: Vectors

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : a -> Vect k a -> Vect (S k) a
```

This declares a **family of types**.
Vect is indexed over Nat and parametrised by Type.

Following the style of Haskell with GADT support,
we explicitly state the type of the type constructors:
Vect takes a Nat and a type as an argument, where Type stands for the type of types.
Each data constructor targets a different part of the family of types.

Function on Vectors - You do not need to test everything

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append []        ys = ys
append (x :: xs) ys = x :: append xs ys
```

The type checkers guarantees that the invariants we might expect by the standard semantics of an append function always hold by **performing simple arithmetical computations at static time**.

Exploiting the knowledge given from expressive types

Because terms of type Vect carry their length as part of their type, the type checker has additional knowledge that it can use to check that operations are implemented and used correctly.

If we wish to look up an element in a Vect you can know at compile time that the location cannot be out of bounds when the program is run.

The type of finite sets

To implement a type safe lookup we can define the type Fin.

```
data Fin : Nat -> Type where
  FZ : Fin (S k)
  FS : Fin k -> Fin (S k)
```

This is the dependent type of finite sets in the sense of "the canonical set of n unnamed elements".

Effectively, it is a type that captures integers that fall into the range from 0 to (n - 1).

Using Fin to access a Vect

Now we can use Fin to define a type-safe indexing operation into a vector, where only values less than the length of the vector are allowed as indices:

```
index : Fin n -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Note that we do not even need to use Maybe in the return type since we are guaranteed a result.

Moreover the type Fin Z is empty so that it is impossible to construct a natural number less than zero. The possibility of indexing into an empty vector falls out of this construction.

Dependent pairs use cases

In particular when dealing with user input, there will often be some properties of the data that we cannot know until runtime.

The length of a vector is one example: once you have read the vector, you know its length, and from there you can check it and reason about how it relates to other data.

The generic solution is provided by dependent pairs.

Inferring Vect length via dependent pairs

```
readVect : IO (len ** Vect len String)
readVect = do x <- getLine
              if (x == "") then pure (_ ** [])
              else do (_ ** xs) <- readVect
                      pure (_ ** x :: xs)

zipInputs : IO ()
zipInputs = do putStrLn "Enter first vector"
               (len1 ** vec1) <- readVect
               putStrLn "Enter second vector"
               (len2 ** vec2) <- readVect
               case exactLength len1 vec2 of
                 Nothing    => putStrLn "Different lengths"
                 Just vec2' => printLn (zip vec1 vec2')
```

a bug

References

1. Edwin Brady - Type-Driven Development with Idris (Manning, 2017)
2. Nordström, Petersson, Smith - Programming in Martin-Löf's Type Theory

Thanks for the attention!

Any question?