Linear Dependent Types for Domain Specific Program Analysis

Marco Gaboardi

Università di Bologna - INRIA project Focus - University of Pennsylvania
gaboardi@cs.unibo.it

Proposal for a 45-60 minutes tutorial based on the works [1],[2],[3] and [4]

Abstract. In this tutorial I will present how a combination of linear and dependent type can be useful to describe different properties about higher order programs. Linear types have been proved particularly useful to express properties of functions; dependent types are useful to describe the behavior of the program in terms of its control flow. This two ideas fits together well when one is interested in analyze properties of functions depending on the control flow of the program. I will present these ideas with example taken by complexity analysis and sensitivity analysis. I will conclude the tutorial by arguing about the generality of this approach.

Higher order functional languages provide a powerful abstraction mechanism helpful both to structure programs in a modular way and to provide a strong semantics ground to programs. However, this same abstraction mechanism makes the analysis of programs more difficult. An invaluable tool to ensure properties about higher order programs are type system. A typing judgment `|- P : A -> B` gives information about the function that the program `P` represents, e.g it tells us that the program `P` represents a function that given as input an object of type A provide as output an object of type B.

Nowadays, there is a full scale of type systems that permit to describe different properties about program. At one end of the scale, there are *simple types* systems. Simple types, like the ones in the example above, provide useful information about the input-output domains of the program and are helpful to ensure *weak* properties of programs like that programs *cannot go wrong*. At the other end of the scale, there are systems like intuitionistic type theories, the type theories behind tools like Coq and Agda, that instead are able to describe very precise specifications of all the aspects of a program and are helpful to ensure also *strong* properties of programs. The strength of these systems rely on a combination of different abstraction mechanisms like dependent types, polymorphism, inductive and coinductive types and universes. The downside of this approach is that the programmer needs to provide an explicit (partial) proof that his program satisfy the intended property. For strong properties this task can be overwhelming.

In between the two ends of the scale there is a full range of other type systems ideas that have been proved to be useful to analyse programs. In this tutorial, I want to present some experience in combining two of these ideas: dependent types and linear types. From our experience, type systems combining these two ideas gives analyses that are useful to prove specific properties of programs.

The key idea of dependent type systems is to let the types depend on the value of terms. A particularly simple example of dependent types is the one represented by *indexed* types [5], [6]. An indexed type `A[I]` can be seen as the type of elements of type `A` that meet the property `I`. In general, `I` can be seen as a boolean predicate even if in some cases it is more convenient to think to it as a property that identifies one element of `A` (singleton types). This kind of indexed types are useful to describe more refined properties of programs. For instance, a typing judgment like

```
i,j : length |-
      append : listChar[i] -> listChar[j] -> listChar[i+j]
```

says that `append` represents a function that takes in input a first list of character of length `i`, a second list of character of length `j` and returns a list of character of length `i+j`. Indexed type systems are very useful to describe the input-output behavior of programs, however when one is interested in describing the properties of functions this approach has some drawbacks.

As an example of this fact, let's consider *function sensitivity*. The sensitivity of a function $f(x)$ is an upper bound on how much $f(x)$ can change in response to a change to x—in other words, if f has sensitivity k, then $|f(x+\delta) - f(x)| \leq k \cdot |\delta|$ for all x and δ. This property, also known as *Lipschitz continuity*, can be extended to entire programs with multiple inputs, and it has important applications in many parts of computer science, including control theory, dynamic systems, program analysis, and data privacy.

Indexed types, can be used to ensure function sensitivity, e.g. a typing judgment as

```
i : real |- P : R[i] -> R[2*i+1]
```

can ensure that the function computed by `P` is 2-sensitive. The way we can deduce this is by looking at the fact that the function f computed by `P` is such that $f(x) = 2 * x + 1$ and by knowing that f is 2 sensitive. This kind of reasoning has some drawbacks: first, the reasoning on the sensitivity is external to the type system. i.e. this is performed on the semantics of the program; second, to capture the precise input-output behavior, the types have to be very rich—this makes difficult to perform the analysis in an automatic way; third, to generalize the analysis to study the sensitivity on function spaces—useful for programs like `map`, `fold`, etc.—we need to extend indexed types to functions. One possible solution is to consider full dependent type systems, like intuitionistic type theory. However, this approach once again makes difficult to perform the analysis in an automatic way.

An alternative is to use other type ideas specifically developed to talk about properties of functions. One important example is represented by *linear types*. One of the main idea of linear logic—from which linear types systems are inspired— is to decompose a function type as `A -> B = !A -o B`. A standard description of this decomposition says that `A -o B` can be seen as a the type of a function that uses its argument exactly once, while `!A -o B` can be seen as the type of a function that uses its argument an arbitrary number of times. A less standard description (but immediate from the semantics of linear logic) says that the *linear* function space constructor `-o` is a mapping from an *individual* of the input to an *individual* of the output; while the *modality*

`!` describes how the individuals of the type `A` are combined to obtain the individuals of the type `!A`. The important point here is that now the modality can be used to capture the property of the function space. To provide a finer analysis of this function space, we generalize the modality ! to a set of indexed-modalities $!_I$.

Let's see how we can use this idea to describe function sensitivity. First, we can say that the linear map like `A -o B` represents a 1-sensitive function from `A` to `B`. Second, by taking an index I to be equal to a number r we can say that the modality `!_r A` is the same type as `A` except that the distance between elements in `A` is now scaled by r. By building the type system accordingly to this idea, we can ensure that a term to which we can assign type `!_r A -o B` represents an r-sensitive function from `A` to `B`. As an example, let's assume that + is a 1-sensitive function in its arguments, i.e. `+ : !_1 R -o !_1 R -o R` then we have a judgment

```
|- λx. x + x :!_2 R -o R
```

ensuring that the function `λx. x + x` is 2-sensitive in its argument. This example also shows why this property can be described by linear types. Indeed, the sensitivity of a function depends on the single uses of the input.

A type system built around this idea have been used in [7] as the basic block of a language for differential privacy. This type system provides an efficient analysis tool to infer the sensitivity of higher order programs [4]. Unfortunately, this analysis fails on many important programs. The main problem is that the sensitivity of the program on one input can depends on some other input. Consider as an example an `iter` function that given a function f maps it on an input value k, a number of time specified by a parameter n, i.e. `iter` $n\ f\ k = f^n(k)$. We can assume to have the following type:

```
|- iter : Nat -> (R -> R) -> R -> R
```

The problem here is that the sensitivity of `iter` on k depends on the value of n and on the sensitivity of f, and this kind of dependency cannot be captured by the indexed-modality $!_r$ alone.

The natural way to generalize the analysis proposed in [7] is to combine it with indexed types and have a similar set of indexes both in types and in modalities. For instance, we can have a judgment:

```
|- iter : Nat[i] -> (!_r R -o R) -> !_(r*i) R -o R
```

saying that `iter` is `r*i` sensitive on its argument, where `i` is the size of n and r is the sensitivity of f and where we use the standard type `->` to omit the sensitivities of the other arguments. A type system able to perform this kind of analysis has been used in [3] to extend the language for differential privacy described in [7]. The resulting system besides combining the indexed type approach with the one of linear types has also some other features like subtyping and quantifiers. The gain in using this generalized approach is the ability to analyse more general programs where the property of interest depends on the control flow of the program.

An important motivation for following the approach described above is given by type inference and type checking. Type checking and inference for linear dependent types can be seen as an extension of the usual ML type checking and inference by two extra phase where the constraint generation and the constraint resolution on the index language are performed. This suggest clearly, a two step approach where constraints are first generated by a standard algorithm and then are passed to an automatic solver. In this way, the strength of the analysis is reduced to the strength of the solver in deciding the generated constraints.

The idea of combining linear and dependent types, dubbed naturally *linear dependent* types, has been originally proposed by myself and Dal Lago to perform complexity analysis of higher order programs [1][2]. Indeed, an analysis for the complexity of programs need to be able to express it as a function of the input values. Moreover, every precise description of the complexity has to consider the control flow of the program. These two considerations motivated the combination of linear types, already extensively used in the area of Implicit Computational Complexity, with indexed types. The kind of types used in [1] are more general than the one I outlined above. Indeed, in [1] a modal type has the shape $!_{a<I}A$. So, it is indexed not only by an index I (representing a natural number) but instead by an inequality $a < I$ that says that the variable a, that can appear in A can assume values less that I. More precisely, for complexity we can think to the type $!_{a<I}A$ as representing the type $A[0/a] \otimes A[1/a] \otimes \cdots \otimes A[I-1/a]$. This generalized form of modality permits to have the value of some modality to depend also on values of other modalities. The system presented in [1] has also other components that make it non-standard. In particular, the system is parametrized on an equational program providing the semantics of the indexes. All those components are needed to obtain a *relative completeness* result stating that the complexity analysis is complete, assuming an oracle to decide the constraints.

The complexity analysis described in [1] is in term of a call by name Krivine-like abstract machine. Dal Lago and Petit [8] have shown how the same approach can be used to give precise information also for a call by value CEK-like abstract machine. Moreover, in [9] they also have shown how the relative completeness result of the type system in [1] can be transferred to a relative completeness result of the type inference process.

The pattern described above is not limited to *sensitivity* and *complexity*. At the end of the tutorial I will show how this idea can be applied also to design type systems permitting other analysis like information flow. The important foundational aspect of this approach is that the operation involved in the constraints on the indexes correspond to the operations needed to reason about the property of interest, and that these operations in turn correspond to different interpretations of the laws of Linear Logic (contraction, weakening, digging and dereliction).

References

1. U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In LICS '11, pages 133–142. IEEE Computer Society, 2011.
2. U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In LMCS, Special Issue of LICS'11, Volume 8(4) 12.
3. M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM POPL 2013*, Jan. 2013.

4. L. D'Antoni, M. Gaboardi, E. J. Gallego Arias, A. Haeberlen, and B. C. Pierce. Type-based sensitivity analysis. *Submitted*, Apr. 2013.
5. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. POPL*, 1999.
6. C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. ICFP*, pages 66–77, 2005.
7. Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, September 2010.
8. U. Dal Lago and B. Petit. Linear dependent types in a call-by-value scenario. In *ACM PPDP 2012*, pages 115–126, 2012.
9. U. Dal Lago and B. Petit. The geometry of types. In *ACM POPL 2013*, 2013.