Dependent types

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

Basics

Dependent types: basics

What can depend on what?
(e.g. what can appear as an argument in an application?)

Simple types

No dependencies involving types
(all types are global)

$\mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$

$\lambda x{::}\, \mathbb{N}.\lambda f{:}(\mathbb{N} \to \mathbb{N}).f\,x$

Terms depend on terms

Polymorphism

$\forall\beta.(\forall\alpha.\alpha) \to \beta$

$\Lambda\beta.\lambda x{:}(\forall\alpha.\alpha).x[\beta]$

Terms depend on types

Dependent types

Types depend on terms

$\Pi(n{:}\mathbb{N}).\mathsf{Vec}\ n \to \mathsf{Vec}\ n$

$\lambda n{:}\mathbb{N}.\lambda v{:}\mathsf{Vec}\ \mathrm{n}.v$

The Curry-Howard correspondence

Correspondence between simply-typed language and propositional logics:

$$\begin{array}{rcll} A \to B & \simeq & A \supset B & \text{(functions and implications)} \\ A \times B & \simeq & A \wedge B & \text{(products and conjunctions)} \\ A + B & \simeq & A \wedge B & \text{(sums and disjunctions)} \end{array}$$

Correspondence between dependently-typed languages and predicate logics:

$$\begin{array}{rcll} (x : A) \to B & \simeq & \forall (x : A).B & \text{(functions and universal quantification)} \\ \Sigma (x : A).B & \simeq & \exists (x : A).B & \text{(dependent pairs and existential quantification)} \end{array}$$

How should we **start** to design a dependently-typed language**?**

Foundation for constructive mathematics (Martin-Löf Type Theory)

Lambda calculus with fancy types (Calculus of Constructions)

With dependent types we can form **types from terms**.
Parameterise B by a term of type A:

$$\Pi(x : A).B(x)$$

Key Q: when are two types equal? (essential for type checking!)

Is $B(2+2)$ equal to $B(4)$?

Determining equality typically requires **normalization** (i.e. computation).

(Separate question: what equalities can we **prove**?)

Pattern matching

Pattern matching with simple types

Simple branching reveals nothing to the type checker:

```
append xs ys = if empty xs then ys
               else cons (head x) (append (tail xs) ys)
```

Either branch can access the head and tail.

Pattern matching exposes the value structure to the type checker:

```
append  Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Only the `Cons` branch can access the head and tail.

Inductive families support **indexing data types by terms**:

an inductive family, Vect:

```
data Vect : ℕ → Type → Type where
  Nil  : Vect Z a
  Cons : a → Vect n a → Vect (S n) a
```

a function vappend over Vect:

```
vappend : Vect m a → Vect n a → Vect (m + n) a
vappend  Nil         ys = ys
vappend (Cons x xs) ys = Cons x (vappend xs ys)
```

the full type of vappend:

```
vappend: {a : Type} → {m : ℕ} →{n : ℕ} →
         Vect m a → Vect n a → Vect (m + n) a
```

Dependent matching may reveal something about *another value*:

```
vappend : Vect m a → Vect n a → Vect (m + n) a
vappend  Nil          ys = ys
vappend (Cons x xs) ys = Cons x (vappend xs ys)
```

1. Matching the first vector with `Nil` tells us that `m` $\equiv$ `Z` in the first branch
2. so the return type in the first branch is `Vect (Z + n) a` $\rightsquigarrow$ `Vect n a`
3. so `ys` has the appropriate type in the first branch

Dependent matching may reveal something about *another value*:

```
zip : Vect n a → Vect n b → Vect n (a,b)
zip    Nil     ys = ?
```

1. Matching the first vector with `Nil` tells us that `n ≡ Z`
2. so the type of `ys` is `Vect Z b`
3. and so `Nil` is the only possible constructor for `ys`

Recursion

Ideally: all functions terminate.

Non-terminating functions can introduce logical inconsistency, e.g.:

```
circular : ∀ (A : Type) → A
circular a = circular a
```

or:

```
data Empty : Type where
  -- (no constructors)

loopy : Empty
loopy = loopy
```

Approximating termination

Problem: termination is undecidable, so we must approximate syntactically

Question: what to do with functions that are not structurally decreasing?

structurally decreasing:

```
length : List a → Int
length []     = 0
length (x:xs) = 1 + length xs
```

not (obviously) structurally decreasing:

```
quicksort :: List N → List N
quicksort [] = []
quicksort (x:xs) = quicksort (filter (<  x) xs) ++
                   x : quicksort (filter (>= x) xs)
```

Reading

Paper 1: termination

The Size-Change Principle for Program Termination

Chin Soon Lee*
Department of Computer Science and Software Engineering
The University of Western Australia
Nedlands 6907
Western Australia
leecs@cs.uwa.edu.au

Neil D. Jones
Datalogisk Institut
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
Denmark
neil@diku.dk

Amir M. Ben-Amram
Academic College of Tel-Aviv–Yaffo
4 Antokolsky Street
Tel-Aviv 64044
Israel
amirben@mta.ac.il

ABSTRACT

The "size-change termination" principle for a first-order functional language with well-founded data is: a program terminates on all inputs if *every infinite call sequence* (following program control flow) would cause an infinite descent in some data values.

Size-change analysis is based only on local approximations to parameter size changes derivable from program syntax. The set of infinite call sequences that follow program flow and can be recognized as causing infinite descent is an ω-regular set, representable by a Büchi automaton. Algorithms for such automata can be used to decide size-change termination. We also give a direct algorithm operating on "size-change graphs" (without the passage to automata).

Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexical orders (also called lexicographic orders), indirect function calls and permuted arguments (descent that is not *in-situ*) are all handled *automatically and without special treatment*, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time.

We establish the problem's *intrinsic complexity*. This turns out to be surprisingly high, complete for PSPACE, in spite of the simplicity of the principle. PSPACE hardness is proved by a reduction from Boolean program termination. An interesting consequence: the same hardness result applies to many other analyses found in the termination and quasi-termination literature.

*This research was done while visiting DIKU.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.4 [**Programming Languages**]: Processors; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

Keywords

Termination, program analysis, omega automaton, PSPACE-completeness, partial evaluation.

1. INTRODUCTION
1.1 Motivation

There are many reasons to study automatic methods to prove program termination, including:

- Program verification: typically deductive methods are used to show partial correctness (the input-output specification is satisfied provided the program terminates), followed by a separate proof of termination [11].
- Automatic program manipulation: termination has to be ensurable when dealing with machine-generated programs, or ones imported from a possibly untrustworthy context.
- Broad interest: termination has been studied in fields including functional programming [8], logic programming [7, 15, 17, 19, 14], term rewriting systems [3, 20] and partial evaluation. Discussion of related work appears at the end of this paper.
- Interesting analysis: termination is not just an "abstract interpretation" of program values, but rather more subtle.
- Use in *partial evaluation*: this is a step towards a binding-time analysis that will guarantee termination of program specialization [12, 2, 9, 10] and still allow an acceptably high degree of specialization in an offline partial evaluator such as Similix [5].

We emphasize here a careful and precise formulation of a simple but powerful *principle* to decide termination. It is owing to this clear statement of the termination criterion

"[A] program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values."

"The set of infinite all sequences that follow program flow and can be recognized as causing infinite descent is an ω-regular set, representable by a Büchi automaton"

"There are many reasons to study automatic methods to prove program termination, including: Program verification [...] Interesting analysis: termination is not just an "abstract interpretation" [...] Use in partial evaluation"

Paper 2: Idris

IDRIS — Systems Programming Meets Full Dependent Types

Edwin C. Brady

School of Computer Science, University of St Andrews, St Andrews, Scotland.

Email: eb@cs.st-andrews.ac.uk

Abstract

Dependent types have emerged in recent years as a promising approach to ensuring program correctness. However, existing dependently typed languages such as Agda and Coq work at a very high level of abstraction, making it difficult to map verified programs to suitably efficient executable code. This is particularly problematic for programs which work with bit level data, e.g. network packet processing, binary file formats or operating system services. Such programs, being fundamental to the operation of computers in general, may stand to benefit significantly from program verification techniques. This paper describes the use of a dependently typed programming language, IDRIS, for specifying and verifying properties of low-level *systems* programs, taking network packet processing as an extended example. We give an overview of the distinctive features of IDRIS which allow it to interact with external systems code, with precise types. Furthermore, we show how to integrate tactic scripts and plugin decision procedures to reduce the burden of proof on application developers. The ideas we present are readily adaptable to languages with related type systems.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) Languages; C.2.2 [*Computer-Communication Networks*]: Network Protocols—Protocol Verification

General Terms Languages, Verification

Keywords Dependent Types, Data Description

1. Introduction

Systems software, such as an operating system or a network stack, underlies everything we do on a computer, whether that computer is a desktop machine, a server, a mobile phone, or any embedded device. It is therefore vital that such software operates correctly in all situations. *Dependent types* have emerged in recent years as a promising approach to ensuring the correctness of software, with high level verification tools such as Coq [8] and Agda [25] being used to model and verify a variety of programs including domain-specific languages (DSLs) [26], parsers [9], compilers [16] and algorithms [34]. However, since these tools operate at a high level of abstraction, it can be difficult to map verified programs to efficient low level code. For example, Oury and Swierstra's data description language [26] works with a list of bits to describe file formats precisely, but it does not attempt to store concrete data compactly or efficiently.

This paper explores dependent type based program verification techniques for *systems* programming, using the IDRIS programming language. We give an overview of IDRIS, describing in particular the key features which distinguish it from other related languages and give an extended example of the kind of program which stands to benefit from type-based program verification techniques. Our example is a data description language influenced by PADS [19] and PACKETTYPES [22]. This language is an embedded domain-specific language (EDSL) [14] — that is, it is implemented by embedding in a host language, exploiting the host's parser, type system and code generator. In this EDSL, we can describe data formats at the bit level, as well as express constraints on the data. We implement operations for converting data between high level IDRIS data types and bit level data, using a foreign function interface which gives IDRIS types to C functions. This language has a serious motivation: we would like to implement verified, efficient network protocols [1]. Therefore we show two packet formats as examples: Internet Control Message Protocol (ICMP) packets, and Internet Protocol (IP) headers.

1.1 Contributions

The main contribution of this paper is to demonstrate that a high level dependently typed language is capable of implementing and verifying code at a low level. We achieve this in the following specific ways:

- We describe the distinctive features of IDRIS which allow integration of low level systems programming constructs with higher level programs verified by type checking (Section 2).
- We show how an effective Foreign Function Interface can be embedded in a dependently typed language (Section 2.6).
- We introduce a serious systems application where a programming language meets program verification, and implement it fully: a binary data description language, which we use to describe ICMP and IP headers precisely, expressing the data layout and constraints on that data (Section 3).

We show how to tackle some of the awkward problems which can arise in practice when implementing a dependently typed *application*. These problems include:

- Dealing with foreign functions which may have more specific inputs and outputs than their C types might suggest — e.g. we might know that an integer may lie within a specific range.
- Satisfying proof obligations which arise due to giving data and functions precise types. As far as possible, we would like proof obligations to be solved automatically, and proof requirements should not interfere with a program's *readability*.

PLPV'11, January 29, 2011, Austin, Texas, USA.

"This paper describes the use of a dependently typed programming language, IDRIS, for specifying and verifying properties of low-level systems programs, taking network packet processing as an extended example."

"Our motivation is the need for systems software verification — programs such as operating systems, device drivers and network protocol implementations which are required for the correct operation of a computer system. Therefore it is important to consider not only how to verify software, but also how to do so without compromising on efficiency, and how to interoperate with concrete data as it is represented in the machine or on a network wire"

Paper 3: Epigram

Why Dependent Types Matter

Thorsten Altenkirch Conor McBride
The University of Nottingham
{txa,ctm}@cs.nott.ac.uk

James McKinna
The University of St Andrews
james.mckinna@st-andrews.ac.uk

Abstract

We exhibit the rationale behind the design of Epigram, a dependently typed programming language and interactive program development system, using refinements of a well known program—merge sort—as a running example. We discuss its relationship with other proposals to introduce aspects of dependent types into functional programming languages and sketch some topics for further work in this area.

1. Introduction

Types matter. That's what they're for—to classify data with respect to criteria which matter: how they should be stored in memory, whether they can be safely passed as inputs to a given operation, even who is allowed to see them. Dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data. As such, they enable you to express more of what matters about data. While conventional type systems allow us to validate our programs with respect to a fixed set of criteria, dependent types are much more flexible, they realize a continuum of precision from the basic assertions we are used to expect from types up to a complete specification of the program's behaviour. It is the programmer's choice to what degree he wants to exploit the expressiveness of such a powerful type discipline. While the price for formally certified software may be high, it is good to know that we can pay it in installments and that we are free to decide how far we want to go. Dependent types reduce certification to type checking, hence they provide a means to convince others that the assertions we make about our programs are correct. Dependently typed programs are, by their nature, proof carrying code [NL96, HST+03].

Functional programmers have started to incorporate many aspects of dependent types into novel type systems using *generalized algebraic data types* and *singleton types*. Indeed, we share Sheard's vision [She04] of closing the *semantic gap* between programs and their properties. While Sheard's language Ωmega approaches this goal by an evolutionary step from current functional languages like Haskell, we are proposing a more radical departure with Epigram, exploiting what we have learnt from proof development tools like LEGO and COQ.

Epigram is a full dependently typed programming language defined by McBride and McKinna [MM04], drawing on experience with the LEGO system. McBride has implemented a prototype which is available together with basic documentation [McB04, McB05] from the Epigram homepage.[1] The prototype implements most of the features discussed in this article, and we are continuing to develop it to close the remaining

"Dependent types [...] provide a means to convince others that the assertions we make about our programs are correct. Dependently typed programs are, by their nature, proof carrying code."

"Epigram can also typecheck and evaluate incomplete programs with unfinished sections sitting in *sheds*, `[ · · · ]`, where the typechecker is forbidden to tread."

"Exploiting the expressivity of dependent types in a practicable way involves a wide range of challenges in the development of the theory, the design of language, the engineering of tools and the pragmatics of programming."

Writing suggestions

Termination
Is termination-checking practical for real-world programs?

Efficiency
Are dependent types an impediment or an aid to efficiency?

Re-thinking
How might dependent types change the way we think about programming?

Radicalism
Do dependent types require radically new ways of programming?

Adoption
What might impede adoption of dependently-typed languages?