Recent Topics in Compilers and Programming Languages

Dependent Types and Automatic Vectorization and Parallelization

Dario Gjorgjevski[1] Petar Tonkovikj[1]
{gjorgjevski.dario,tonkovikj.petar}@students.finki.ukim.mk

[1]Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University in Skopje

May 7, 2017

Contents

Compilers

Not so modern, even today?

Even today, the architecture of a compiler is the same as it was more than thirty years ago, when the modern foundations were laid out. Nonetheless, that is only the big picture – the difference is in the little things:

- Functional languages are gaining traction. Type theory and λ-calculi are being researched more than ever.
- Compilers are becoming smarter:
  - JIT and dynamic optimizations are extended to static languages.
  - Compilers help in tackling multi-core programming.
  - Compilers are (partially) responsible for memory safety.

Today's topics

We have researched two topics: one related to functional programming and type theory, and the other to code optimization.

Dependent types Types whose definition depends on a value.

Exploiting parallelism Automatic vectorization and parallelization.

We believe that these topics are very relevant to today's trends in compilers and programming languages, and will try to introduce you to them.

Contents

Dependent types

Motivation

Types matter. They allow us to classify data:

- how to store them in memory;
- whether they can safely be passed to an operation; and
- who is allowed to see them.

Dependent types are expressed in terms of data, explicitly relating their inhabitants to the data.

Dependent types

Motivating example

Consider the typical `head` function.

```
head :: a list -> a
head (x : xs) = x
head [] = error "no head"
```

Program correctness

Is `head z` a correct program? A traditional (e.g., Haskell's) type checker can't tell.

Dependent types

Fixing things

Let us create a datatype that tracks the length at compile-time.

```
data Nat = 0 | Succ Nat
data Vec (a : *) (n : Nat) where
  nil  : Vec a 0
  cons : a -> Vec a n -> Vec a (Succ n)

head :: Π (x : Nat). Vec a (Succ x) -> a
head (cons x xs) = x
-- head nil case is impossible!
```

Success!

If `head z` typechecks, it must be correct, i.e., there must be a head.

Formal definition

Definition (Dependent type)

Given a type $A : \mathcal{U}$ in a universe of types $\mathcal{U}$, let $B \colon A \to \mathcal{U}$ be a *family of types* which assigns to each term $x : A$ a type $B(x) : \mathcal{U}$.
A function whose type of return value varies with its argument (i.e., it has no fixed codomain) is called a *dependent type*. It is written as

$$\prod_{(x:A)} B(x).$$

A "pair of integers" is a type. A "pair of integers where the second is greater than the first" is a dependent type because of the dependence on the value.

More applications

Attaching invariants to datatypes

We know that AVL trees are binary search trees balanced according to the AVL criterion. Dependent types allow us to explicitly encode this information.

```
Inductive tree: Type :=
  | Leaf: tree
  | Node: tree -> A -> tree -> tree.

Inductive bst: tree -> Prop := ...
  (* to be a binary search tree *)

Inductive avl: tree -> Prop := ...
  (* to be balanced by the AVL criterion *)
```

More applications

Attaching invariants to datatypes

Proving that base operations preserve the `bst` and `avl` invariants is sufficient to prove that any operations built on top of them are correct.

```
Definition add (x: A) (t: tree) : tree := ...
  (* adds a new element to the tree *)

Lemma add_invariant:
  forall x t, bst t /\ avl t ->
    bst (add x t) /\ avl (add x t).
```

Simply-typed λ-calculus

$\lambda_{\rightarrow}$ for short

- $\lambda_{\rightarrow}$ is the "smallest" statically typed functional language.
- Every term is explicitly typed and no type inference is performed.
- There are only base types (no type variables) and hence no type polymorphism.

We will afterwards extend the rule of $\lambda_{\rightarrow}$ to include dependent types.

Simply-typed λ-calculus

Abstract syntax

The language $\lambda_{\to}$ consists of only two constructs: types and terms.

$$\begin{array}{rcll} \tau & := & \alpha & \text{base type} \\ & | & \tau \to \tau' & \text{compound type} \end{array}$$

α is a base type, and $\tau \to \tau'$ is a compound type denoting a function from τ to τ'.

$$\begin{array}{rcll} e & := & e : \tau & \text{annotated term} \\ & | & x & \text{variable} \\ & | & e\ e' & \text{application} \\ & | & \lambda x \to e & \text{lambda abstraction} \end{array}$$

Simply-typed λ-calculus

Type system

Type rules are of the form $\Gamma \vdash e : t$, indicating that term e is of type t in context Γ. We denote by $*$ the set of base types.
We need the following type rules:

- (ANN) to check annotated terms against their type annotation, and then return the type;
- (VAR) to look up the type of a variable in the environment;
- (APP) to check the type of function arguments and return the type of the range as result;
- (CHK) to check if inferred and specified types are the same; and
- (LAM) to check the body of a lambda abstraction in an extended context.

Dependently-typed λ-calculus

λ_Π for short

Adding dependent types to $\lambda_\rightarrow$ makes things simpler in some, and more complicated in other aspects.

- Values can now appear in types, so there is no longer a syntactic distinction between types and terms.
- Types must be evaluated, too.

We move away from the notation in def. 1, and write $\forall x : A.\, B(x)$ instead. $\forall$ is called the *dependent function space*.

Dependently-typed λ-calculus

Abstract syntax

Integrating all constructs in the term language, we obtain a somewhat simpler syntax:

$$
\begin{array}{rcll}
e, \rho, \kappa & := & e : \rho & \text{annotated term} \\
& | & * & \text{the type of types} \\
& | & \forall x : \rho.\, \rho' & \text{dependent function space} \\
& | & x & \text{variable} \\
& | & e\ e' & \text{application} \\
& | & \lambda x \to e & \text{lambda abstraction}
\end{array}
$$

ρ and κ denote types and kinds, respectively. The kind $*$ is now a term, as expected.

Dependently-typed λ-calculus

Type system

Contexts no longer check well-formedness, only store evaluated types. The difference in type rules is as follows:

- (ANN) now does not check well-formedness, but performs type checking instead; also, since ρ might not be a value, it is first evaluated;
- (STAR) is a new axiom saying that the kind of $*$ is $*$ itself;
- (PI) is a new rule for the dependent function space: both ρ and ρ' must be of kind $*$, and ρ' can refer to x;
- (CHK) works as before, but now types are evaluated;
- (APP) applies dependent function types $\forall x : \tau.\, \tau'$, and lets τ' refer to x when being evaluated.

Interpreter for λ_Π

...and a possibility for a compiler

Translating the syntax, evaluation, and typing rules given in a language like Haskell is not hard. Löh, McBride, and Swierstra [4] have a great report on that matter.
However, compilers are trickier, because type-checking requires evaluation itself. Possible solutions:

- Implement a virtual machine which compiles terms to bytecode on the fly and performs conversion checks [2];
- Generate code in a language which, upon execution, checks all conversions needed to type-check a dependently-typed program [1];
- Enforce terms appearing in types and terms intended to be run to be parts of two disjoint languages. Then, compile the "type-level" code first and execute it when checking the types of "term-level" code.

Contents

Introduction

Programmers have always had to adapt to concurrent paradigms, but more and more work has been delegated to compilers themselves.

- Parallelization is hectic and error-prone when done by hand.
- Code that exploits parallelism can be automatically generated.
- The simple cases cover most of the real use.

How can compilers help?

There are two approaches: automatic vectorization and automatic parallelization.

Automatic vectorization

Definition (Vectorization)

Vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (series of adjacent values).

Automatic vectorization [5] has the compiler do all of this automatically – without any programmer intervention.

Single instruction, multiple data

SIMD for short

SIMD is an instruction set [3] which:

- Allows data-level parallelism; but
- Does not allow concurrency.

In other words, each instruction uses additional registers to provide vectorization.

Single instruction, multiple data

An example

```
for (i = 0; i <= MAX; i++)
    c[i] = a[i] + b[i];
```

Now, consider the two figures:

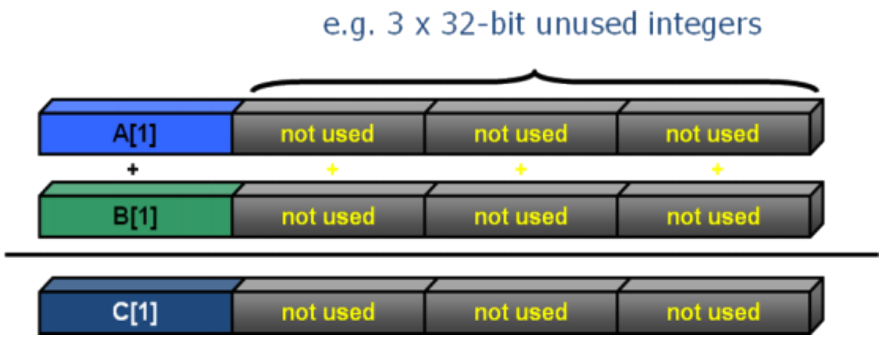

Figure: Disabled vectorization.

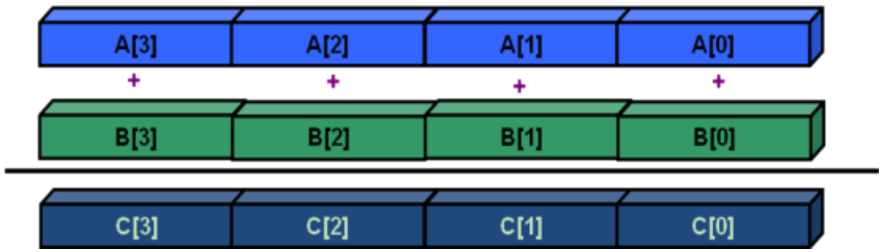

Figure: Enabled vectorization.

Vectorization criteria

Which loops can be vectorized?

Countable The loop trip count must be known at entry to the loop at runtime and must not be data-dependent.

Single entry and single exit The loop must not contain break commands.

Straight-line code Branches are permitted only if they can be implemented as masked assignments.

The innermost loop of a nest Only the innermost loop of nested loops can be vectorized, unless an outer loop is made the innermost loop by a different optimization method.

No function calls The loop must not contain function calls, unless they are inline functions (in order to have straight-line code).

Obstacles to vectorization

Non-contiguous memory access

If the variables are not adjacent in memory, there is no point in vectorization as they can't be operated upon in a single instruction.

```
// arrays accessed with stride 2
for (i = 0; i < SIZE; i += 2)
    b[i] += a[i] * x[i];

// inner loop accesses a with stride SIZE
for (j = 0; j < SIZE; j++) {
    for (i = 0; i < SIZE; i++)
        b[i] += a[i][j] * x[j];
}

// indirect addressing of x using index array
for (i = 0; i < SIZE; i++)
    b[i] += a[i] * x[index[i]];
```

Obstacles to vectorization

Data dependencies

Vectorization entails changes in the order of operations $\Longrightarrow$ there must not be any data dependencies.

```
// read-after-write
A[0] = 0;
for (j = 1; j < MAX; j++)
    A[j] = A[j - 1] + 1;

// write-after-read
for (j = 1; j < MAX; j++)
    A[j - 1] = A[j] + 1;
```

Automatic parallelization

- Vectorization is useful when we perform a single operation on lots of data.
- However, some code cannot be vectorized, but can still be divided into independent chunks.
- These chunks can be run in parallel $\implies$ have the compiler detect loops which can be parallelized.

This is a *concurrent* approach.

Parallelization criteria

Which loops can be parallelized?

- The number of iterations must be known before entry into a loop so that the work can be divided in advance (while loops cannot be done in parallel).
- There can be no jumps into or out of the loop.
- The loop iterations must be independent.

Contents

Conclusion

We presented and analyzed two recent topics in compilers and programming languages: dependent types and automatic parallelization and vectorization.

- Dependent types found their first uses in proof assistants such as Agda and Coq.
- The Idris language uses dependent types for general-purpose programming.
- Automatic parallelization and vectorization are “The Holy Grail” of compiler optimizations. However, they are currently limited to simple blocks of code, and many open problems remain.

References I


Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. "Full Reduction at Full Throttle". In: *Proceedings of the First International Conference on Certified Programs and Proofs*. CPP'11. Kenting, Taiwan: Springer-Verlag, 2011, pp. 362–377.


Benjamin Grégoire and Xavier Leroy. "A Compiled Implementation of Strong Reduction". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 235–246.


John Hennessy and David Patterson. *Computer Architecture. A Quantitative Approach*. 5th ed. Morgan Kaufmann, Sept. 16, 2011.


Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus". In: *Fundam. Inf.* 102.2 (Apr. 2010), pp. 177–207.

References II

Mark Sabahi. *A Guide to Auto-Vectorization with Intel® C++ Compilers*. Apr. 27, 2012. https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers.