Language Independent Dependently Typed Core

Tamar Christina
Dept. Information and Computing Science,
Utrecht University,
Research Proposal (Pre-Draft)
T.Christina@students.uu.nl

December 1, 2010

1 Motivation

Every year fortunes are spent in testing programs for correctness, on creating the tests themselves and on the manpower to run these tests. Projects are routinely late because of programming errors, which in computer science are referred to as *Bugs*. They not only indicate a program crash; any behavior of a program of function which is not specified or conforming to a predetermined specification may lead to unexpected and even dangerous results. These errors cost more the later they are found in the development lifecycle[1], in fact an error that costs €1000,- to correct during the programming phase may easily cost €5000,- when uncovered during the testing phase. Once caught when the software has gone in production they can end up costing up to 25x times the amount it would have to fix early on, and sometimes the costs are even so high that correction is no longer possible.

It doesn't help that programs are becoming more and more complex and that the time allotted to develop them is becoming shorter. This creates an increasingly large pressure to getting things right the first time. This is where *type systems* come in: a type system provides a means of determining the absence of certain kind of programming errors by statically verifying properties of the program (usually done by a compiler). The compiler rejects ill-typed programs, instead letting them pass and becoming part of actual running code. A type system enforces a contract between a caller and its arguments. An simple example of this is when a programmer writes an expression like `1 / "hello"`. A compiler should reject such a program, stating the fact that `"hello"` is not a number, and since the division operator / requires two numbers this program is cannot be meaningfull.

A characterization of a type system is:

> "A *Type System* is a tractable syntactic method for proving the absence

of certain program behaviors by classifying phrases according to the kinds of values they compute"[6]

This definition implies that type systems imply a form of formal correctness prover, providing a number of important functions:

Safety Rejecting of ill-typed programs at compile time, catching bugs sooner rather than later thus saving time and money.

Optimization Because types are known at compile time, more efficient code can be generated thus making programs run faster.

Documentation The more expressive the type system, the more it says about the function and how to use it and which values to express as a result. This serves as part of the documentation as to the meaning of the function.

Abstraction They allow programmers to think of programs from a much higher level, without worrying about how a particular data structure has been implemented.

Not all type systems are created equally however. Programming languages like Haskell[7] provide very expresive type systems when compared to more conventional programmming languages such as C and C++.

An actual, promising development is de coming to life of *dependently typed* programming languages such as Coq[8] and Agda[9], in which the distiction between the world of types and the world of values disappears: in a dependently typed language any property of a program can be encoded and verified, be it with substantial assistance from the programmer. Because dependently typed type systems are also proof assistants a well-typing of a program is a de-facto proof object of stating the formulated properties.

$$matrix\_multiply : matrix_{(k,m)} \times matrix_{(m,n)} \rightarrow matrix_{(k,n)} \tag{1}$$

It says that the inputs should be matrices, one of size (k,m) and the other (m,n) and that the output of invoking the function on two valid arguments will be a matrix of size (k,n). Not only is it checked that both arguments are matrices (types), but also that the sizes match (based on *run time* values). In order to define such a function one needs to be able to provide sufficient evidence to the type system so it can verify the stated runtime properties.

This flexibility allows us to find a much wider class of errors, without even having to run (i.e. to test) the program: far fewer tests should be written to test the program because we already have a proof of correctness. Furthermore the proof is exhaustive, something which is hard to achieve using testing.

Sadly, dependently typed programming languages are not as main stream as Java or C++ even though they provide much better quality control and static checking

abilities. This is in my opinion due to that they are usually harder to use and learn and the programs they produce are generally slower than those produced in the competing languages. For these two reasons they are often not used even though they provide a higher degree of safety.

2 Research

Despite the fact that there exist many different programming languages, they have a lot in common, which becomes amongst others implies that they can all be mapped onto a common (low-level) code language. *I propose the construction of an* dependently typed core language *to which a wide class of other programming languages can translate to. This provides the same level of safety as is to be expected from a dependently typed language, whereas at the same time this extra available typing information can be exploited to produce applications that perform competitively.*

3 Approach

Along the way to completing the task at hand there are a few milestones, each can stand on its own but they ultimately form a whole.

3.1 Core language development

The first component that needs to be created is the actual language, taking as inspiration TyCore [3] and GHC Core [6]. First a complete specification has to be made. This includes developing both Syntax and Semantics.

Being a dependently typed core language means that it will also have values lifted to the domain of types. The difficulty here is trying to come up with a language that's complete enough to be able to describe rich languages such as Agda and Haskell, but also one that we can gain the most opportunities for optimizations from.

One way to do so is to keep the language as simple as possible without sacrifying expressiveness. This would also make it easier to perform type checking in the language itself in case that has not been done by a front-end compiler. Since we need to have typing information, it is essential to develop a type system for the language, probably by adapting an existing type system to the new core language.

3.2 Strictness Analyzer

In order to perform certain kinds of optimization in order to make the code generated perform competitively a strictness analyzer has to be developed for the language and its constructs.

A strictness analyzer looks at the arguments of a function and determines whether they are strict or non-strict. In terms of higher-order functions this is especially tricky.

3.3 Supercompilation on Core language

One of the most important parts of making a competitive language from a higher level language is how to speed up the execution time of the application. One way to do this is to evaluate as much as possible at compile time. This implies that simplification and evaluation rules have to be developed for the language. They have to be proven safe and terminating, and in the end will have an added side effect of creating more optimization opportunities.

3.4 Calling convention & other optimizations

With the strictness analyzer in place and a simplified AST, the next thing to look at is what other optimizations we can apply. Calling convention optimization is one we can certainly apply. But having a fully developed language means we have many more opportunities to express optimization opportunities.

3.5 Embedding in UHC

In order to evaluate the practicality and real world performance of the backend an implementation for UHC (the Utrecht Haskell Compiler) will be provided. This compiler was designed in such a way as to provide an ideal experimentation environment, since its construction is completely modular (see figure 2) and individual components can be relatively easily be replaced or adjusted. To illustrate where in the pipeline of UHC this would fit in the makeup of UHC should be explained:

The front-end of the UHC compiler reads in Haskell code and converts it to EH *(Essential Haskell)* code. *Essential Haskell* is a de-sugared form of Haskell. The bindings are also arranged in a manner that most actions like type checking can be done in a single top-down pass.

Every compiler performs the first two stages. The first part, the parser converts the *concrete syntax* (e.g. what the user writes down) into *abstract syntax* (what the compiler uses and what the code actually means). After this phase the *abstract*

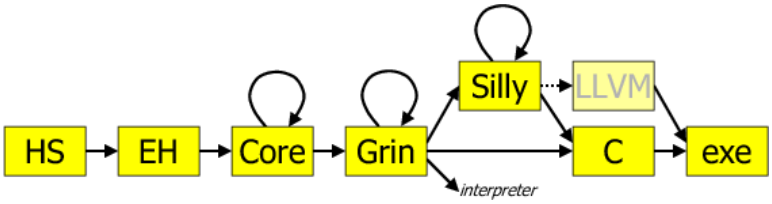

Figure 1: UHC pipeline

Figure 2: the UHC pipeline

syntax is de-sugared and converted into the internal representation that the compiler will continue using. De-sugaring is needed because programming languages provide a lot of convenience syntax for programmers. They make things easier to understand for programmers but they're not needed in terms of the core syntax of the language.

The next three phases: Core, Grin, Silly are all different optimization phases, each having its own syntax and optimization passes. The optimizations get increasingly low level the deeper in the pipeline you go. The next blocks are code generation blocks, where an actual executable gets generated. Figure 4 shows where our our new intermediate language, *TyCore*, would fit into the pipeline.

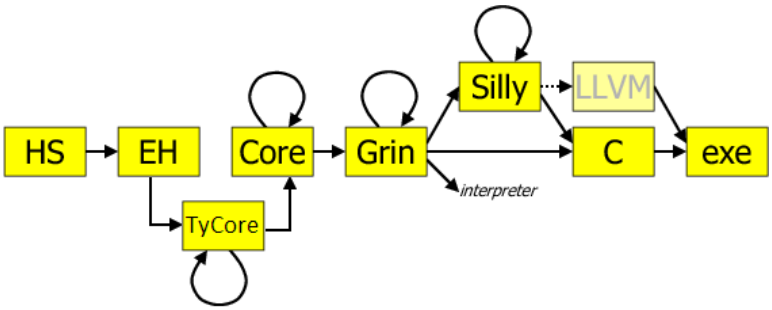

Figure 3: UHC pipeline

Figure 4: The place of *TyCore* in the pipeline

3.6 Evaluatiton & Benchmarks

At the end, to prove that the results are what were claimed, I will provide benchmarks on the "classical" UHC and the one with the developed backend. In order to prove that the language can also optimize other languages we will provide a front end for another language and compare its results as well.

4 Related work

4.1 Types are calling conventions[5]

In this paper Maximilian C. Bolingbroke and Simon Peyton Jones introduce an idea that we can use the arguments of functions themselves as a sort of calling convention. A calling convention describes among other things how arguments should be passed to a function, what to expect as the result of applying a function to some or all arguments, how many arguments should be given to the function etc.

In terms of lazy languages such as Haskell an interesting is how laziness is achieved, if a function always evaluated an expression, it should be possible to instead of passing a thunk to the function that we evaluate it and pass the actual value. This saves us the overhead of passing thunks, and in some instances, of even creating them.

A core language for Haskell is introduced that is based on A-normal form. It is an explicitly typed language[1] which is similar to SystemF but with a slightly more expressive type scheme so that calling conventions can be expressed. It is used in part as inspiration to the next topic.

4.2 Strictness Optimization in a Typed Intermediate Language[4]

In this thesis project Tom Lokhorst has demonstrated that a dependently typed core language can be used for optimizations. It is based on the External GHC core, Henk and strict[2] Core. It is also an explicitly typed Strict language . It supports the passing of multiple arguments at once and the returning of multiple arguments at once from a function. It also has a single representation for values, types and kinds. Kinds are the types of Types. Just as every value has a type, every type also has a type.

To illustrate this, the identity function id is defined as

$$\begin{aligned} Id :< a :_1 * > \ \rightarrow \ < \_ : a > \ \rightarrow \ < \_ : a > \\ = \lambda < a :_1 * >< x : a > \ \rightarrow \ < x > \end{aligned}$$

Because the same syntax is used for both terms and types, the $:_1$ is used to indicate that we're talking about a value. The type of the function about reads as "for a type a with kind *, the function id takes an argument of type a and returns a value of type a" the $<>$ are used to indicate sequences of values. This is how multiple arguments are passed or received.

[1] Explicitly typed means that for every value, terms and expression a type has been given

[2] This means arguments are passed by value (e.g. no thunking)

As the function above demonstrates the types of an argument are also part of the function itself. Which means that when applying the function one of the arguments should be a type argument. An example is id <Int> <3>.

Even though TyCore is strict it can still express laziness. This is done by having explicit notation for laziness which is denoted using {}. For instance <{Int}> means a lazy sequence. In order to force a lazy computation the notation
ensuremathλ is used. E.g.
ensuremathxλ forces the computation of x.

The most obvious and simplest optimization that can be done is *arity raising*. This in effect is just combining multiple arguments from a function into one longer sequence. This is not always possible and should be done with care.

The second optimization is strictness optimization. This is a rather tricky bit, and the more expressive the core language the more difficult this becomes. As mentioned before one important thing is to pass as many values values strictly as possible. This saves memory and time. One might wonder why we don't pass all argument strictly. The reason for that is that Haskell (and a few other languages) is a lazy language. Passing all arguments as strict would change the semantics of the language and produce erroneous results. To illustrate this consider the function const. Const takes two values and just returns the 1^{st} value and discards the second one. In a lazy language *const* 1 $(2 / 0)$ would succeed returning 1. But if we were to pass every argument strictly then before const is even called we would get a division by 0 exception due to the evaluation of the second argument.

In order to achieve strictness optimization every argument is annotated with an annotation indicating whether it's strict or not. An argument is strict in a function if it is inspected inside the body of the function. In most functional languages functions are first class citizens, which means we can pass functions as arguments to functions and receive functions as a result of a computation.

If we were to look at the function map :: $(a \to b) \to ([a] \to [b])$, which maps a function from a to b on every element of a list of a's returning a list of b's. Determining whether the arguments of the function passed to *map* is strict or lazy depends on the function that is passed to map. For instance passing $(+1)$ to *map* implies that a is strict, while if we pass *const* 1 then a is lazy.

4.3 A Supercompiler for Core Haskell

This paper by Neil Mitchell and Colin Runciman [3] introduces the concept of super-compilation on a Haskell core language. The concept of super-compilation entails evaluating the program as much as possible at the time of compilation. The idea being that after doing this the resulting residual program would be more efficient and execute faster because it has less work to do compared to the starting

program.

It attacks the problem on 3 fronts:

Deforestation During a long chain of computations, especially with lists, intermediate values are produced and consumed at ever link in the chain as it calculates the results. This is an obvious source of inefficiency as memory has to be reserved, written to and read from for each of these intermediate lists. The idea of deforestation is to change the program in such a way that no intermediate lists are created or needed.

This would speed up computation such as $print \circ length \circ words =\!\ll getContents$ where without this optimization first the content is read in and put in a list, then this lists is split up into lists of lists, then the length is calculated by traversing this list again and finally we can print the length.

Argument passing Functions are first class citizens in functional programming languages. This offers a great amount of flexibility and abstraction but also introduces a source of inefficiency. Consider the function *dropWhile* that expects as a first argument a function which acts as a predicate and determines whether to stop dropping values from the list. This function is written recursively; in each iteration it calls itself passing the same function as an argument. Having to continually pass this argument along is a pretty expensive thing.

Thunks In lazy languages computations are performed as needed. This enabled the ability to defer computing something to a later time. In order to do this there needs to be a way to express and store suspended computations. Such suspended computations are called *thunks*. When a thunk is forced its value is computed and the thunk is then updated with this value. Creating a thunk is not a cheap operation, it costs time and resources. If we think back to deforestation and argument passing it now becomes clear that having to create intermediate lists or repeatedly passing the same argument around is quite expensive. This is handled in this paper by creating specialized functions which have been partially applied to some of its arguments.

The main contributions of this paper are a reasonable stopping criteria for the unfolding of definitions and a strategy for evaluation. They focus all their optimization strategies on only **case**-expressions and **let** bindings.

They go about by it defining a *split* function that given an expression returns the expression with *holes* and a list of child expressions which were in the place of these holes. They then try to optimize the child expressions and finally the parent expression as a whole, having filled in the holes with optimised expressions.

It has been shown that in a dependently typed core language it is both feasible and practical to formulate the optimization needs. However it stops short of defining a

complete formal syntax and semantics for such a language. It proves what can be done, but just scratches the surface on the possibilities.

5 Expected contributions

None of the examples above form a cohesive whole, and most of them stop short of giving a practical specification. They all prove that some parts of what's proposed here is possible, at least to a very specific language.

The contributions I intend to make in this field are:

Provide a language agnostic strongly typed core language
: A core language that can potentially be used in individual compilers as a backend to perform various transformations or optimizations on. Such a language would help further the adoption of dependently typed languages by giving a common language which all compiler writers can work on improving. [4]

Specify and develop a set of optimizations to improve performance
: A set of optimizations to make it possible to generate more efficient code and provide better resource management [5]

Provide type safe program transformations
: A collection of static program transformations to evaluate as much as possible at compile time. And to expose more potential points of optimizations. Based on Supercompilation [3]

Provide an example implementation in the Utrecht Haskell Compiler
: To illustrate that the language is capable of all that's claimed a reference implementation will be provided for the Utrecht Haskell Compiler, along with benchmarks comparing the different compiler/languages.

References

[1] McConnell, Steve *Code Complete. 1st* Redmond, Washington : Microsoft Press, 1993. pp. 25-26. ISBN 1-55615-484-4.

[2] Computer Language Benchmark. [Online] 2010. [Cited: November 24, 2010.] `http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=ghc&lang2=gcc`

[3] Mitchell, Neil and Runciman, Colin. *A Supercompiler for Core Haskell* 2008, IFL, pp. 147-164.

[4] Lokhorst, Tom. *Strictness Optimization in a Typed Intermediate Language.* Utrecht, The Netherlands : Dept. of Information and Computing Sciences, 2010.

[5] Bolingbroke, Maximilian C. and Jones, Simon L. Peyton. *Types Are Calling Conventions.* 2009, ACM.

[6] Benjamin C. Pierce and David N. Turner *Local Type Inference* Indiana University and Technology Transfer Center

[7] A purely functional programming language `Haskell.org`

[8] `http://coq.inria.fr`

[9] `http://wiki.portal.chalmers.se/agda/pmwiki.php`