CHALMERS

Implementing and Optimizing a Simple, Dependently-Typed Language

Master of Science Thesis in the Program CSALL

MICHAEL BLAGUSZEWSKI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2010

Implementing and Optimizing a Simple, Dependently-Typed Language

MICHAEL E. BLAGUSZEWSKI

Examiner: PETER DYBJER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2010

This thesis presents a compiler for the simple functional programming language LambdaPi, which includes dependent types. The compiler is written in Haskell and uses LLVM, a framework for building optimizing compiler backends. It can compile the complete standard library provided by LambdaPi's authors into native machine code. It is not much of an optimizing compiler, but several obvious opportunities for improvement exist.

First I discuss the theoretical background of project: the principles of dependent types and the languages which include them. I also give a brief overview of the LLVM system and of related work. The second section describes the process of implementation, which was done in stages from a trivial calculator language up to full LambdaPi. And finally we consider opportunities for optimization. Some of these stem from Edwin Brady's [2005] analysis of Epigram, while others are lower-level and can be performed for us by LLVM.

1 Background

1.1 Dependent Types

Over the last several decades, statically-typed functional programing languages like OCaml and Haskell have matured and started making inroads into industry. Features like type inference that they pioneered are starting to show up in more conventional languages like Microsoft's F# and the upcoming standard for C++. In the meantime, researchers in this tradition have moved on to consider more powerful and exotic calculi as the basis for programming languages. Part of this has been incremental, such as Haskell adding GADTs and relying on System F_C as the core calculus. But in the last decade a series of languages have taken a more drastic approach of embracing what are known as dependent types.

Whereas languages with polymorphism allows types to depend on and be indexed by other types, dependently-typed languages let types depend on *values*. Different languages take this capability in different directions. Xi's DML [Xi and Pfenning 1998] is an extension of ML with dependently typed arrays. The vector type is indexed both by the type of its contents, and with an integer indicating its length. `a' array 3` and `a' array 4` are completely different types, and DML's compiler can guarantee the absence of array bounds errors.

```
fun reverse(l) =
let
  fun rev(nil, ys) = ys
    | rev(x::xs, ys) = rev(xs, x::ys)
  where rev <| {m:nat) {n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
in
  rev(1, nil)
end
where reverse <| {n:nat) 'a list(n) -> 'a list(n)
```

List reversal function from [Xi and Pfenning 1998]

The use of dependent types in DML is deliberated limited, and the language for index calculations constrained, so that typechecking is reduced to the decidable problem of solving a system of linear constraints.

Cayenne [Augustsson 1998] is a language that incorporates dependent types in a more general way, while still retaining the feel of a conventional programming language. Types are treated as values, and it's possible to write functions to calculate types, and use them in type signatures. The first example Augustsson uses is `printf`:

```
PrintfType :: String -> #
PrintfType (Nil)         = String
PrintfType ('%':('d':cs)) = Int    -> PrintfType cs
PrintfType ('%':('s':cs)) = String -> PrintfType cs
PrintfType ('%':( _ :cs)) =           PrintfType cs
PrintfType ( _ :cs)       =           PrintfType cs

printf' :: (fmt::String) -> String -> PrintfType fmt
printf' (Nil)         out = out
printf' ('%':('d':cs)) out = \ (i::Int)    -> printf' cs (out ++ show i)
printf' ('%':('s':cs)) out = \ (s::String) -> printf' cs (out ++ s)
printf' ('%':( c :cs)) out =                  printf' cs (out ++ c : Nil)
printf' (c:cs)         out =                  printf' cs (out ++ c : Nil)

printf :: (fmt::String) -> PrintfType fmt
printf fmt = printf' fmt Nil
```

Example from [Augustsson 1998]

Typechecking Cayenne involves evaluating arbitrary type-level functions which may not terminate. As a compromise the implementation lets the user specify the maximum number of evaluation steps to perform because aborting the typechecking process.

At the extreme are languages like Agda [Norell 2007] and Epigram, which focus on theorem proving in the typechecker rather than traditional programming. This is due to the Curry-Howard isomorphism which allows type signatures to be construed as logical propositions, and programs as their proofs. Typechecking then acts as a theorem prover, and the development environment as a proof assistant. Some Agda programmers may never run their programs after typechecking. Such languages also lend themselves to a hybrid approach where part of the program does useful work, and other part encodes a proof the useful code's correctness. Thus certain functions are important at runtime, and others only useful at compile time.

Unlike Cayenne, Agda and Epigram are not Turing complete. They allow only functions for which the compiler can easily determine termination, for instance by limiting recursion to primitive recursion on inductively defined datatypes. Nevertheless, it is still possible to perform larger-scale programming tasks in these languages. Oury and Swierstra [2008] present a number of practical examples using Agda. Their finale is a relational database interface that—with the help of a tool to generate types from database schema—turns query validation into a typechecking problem.

1.2 LambdaPi

LambdaPi is a very simple dependently-typed language. It is presented in [Löh, McBride and Swierstra 2001] as an example of how to implement such a language. The authors describe the workings of a LambdaPi typechecker and interpreter in detail and provide source code which has been incorporated into the current work.

As the name implies, LambdaPi is a very minimal implementation of the lambda calculus, with a ∏ operator for dependent types. In LambdaPi syntax this operator is written `forall` to emphasize the parallel with logic, so `forall a::*. a` → a would be the type of the polymorphic identity function. Note how we've used the name of the argument as a variable in the right-hand side of the expression. The symbol `*` represents the type of types, and unlike in some other languages the type of `*` is `*`. Since this is a dependently-typed language, variables introduced with `forall` can refer to any value, not just types.

Operators are provided for function abstraction and application, as well as several built-in types: `Nat`, `Fin`, `Vec` and `Eq`. `Nat` represents natural numbers, `Fin` is a finite subset of the naturals, `Vec` represents vectors indexed by container type and length, and `Eq` is used to state equality when encoding proofs in LambdaPi programs. There are no user-defined product or sum types, a restriction that has major consequences for implementation and optimization.

```
-- addition of natural numbers
let plus =
  natElim
    ( \ _ -> Nat -> Nat )           -- motive
    ( \ n -> n )                    -- case for Zero
    ( \ p rec n -> Succ (rec n) )   -- case for Succ

 let seven = plus 3 4 :: Nat
```
From the LambdaPi Prelude

Like Agda, LambdaPi does not support general recursion. Name bindings are merely for convenience. Non-trivial computations are performed using constructors and eliminator functions. The latter look much like folds in function programming, and indeed `Vec-Elim` is essentially a dependently typed equivalent of Haskell's `foldl`.

The authors provide a basic standard library for LambdaPi, referred to as the Prelude. This contains definitions of basic types such booleans, and builds more advanced constructions like list projection and mapping. It also includes some type-level functions to encode invariants, such as Leibnitz's Law. These features make it a good test case for compilation strategies.

1.3 LLVM

LLVM—the Low Level Virtual Machine—is a framework for building compiler backends [Lattner 2008]. It was starting in 2002 as a research project at the University of Illinois, and has since attracted industry attention. The largest supporter is Apple Inc., which uses it to compile programs for graphic cards, and are developing a compiler for C-based languages to replace GCC.

LLVM provides an intermediate source-level representation (assembly code) and binary representation (bitcode), corresponding to assembly and object code in conventional compilers. This intermediate representation is in single static assignment (SSA) form, for which a great many optimization algorithms are known. LLVM can run these optimization passes both on a per-file basis and during linking of bitcode files, when whole-program strategies are permitted. As a final step it generates native code.

The LLVM tools are open-source, and can be used in two ways. The compiler-design class at Chalmers has students generate LLVM assembly code as text, which then gets run through an external assembler and optimizer. However the preferred method for production compilers is to use an API to generate assembly instructions programmatically. That is the method used in this thesis. The native API of LLVM is written in C++, but I have used a set of Haskell bindings written by Bryan O'Sullivan and Lennart Augustsson.

LLVM assembly code is a typed language. In fact since it is intended for machine generation it requires frequent annotations, sacrificing readability for safety. Explicit type conversions can be performed using the bitcast operator, so it is possible to subvert the type system whenever necessary, at the risk of confusing the optimizer.

1.4 Related Work

Other compilers have been written for functional languages using LLVM. A couple of these have taken the approach of extending an existing compiler's backend, mapping its intermediate code representation into LLVM assembly code. Terei [2009] does this with GHC, translating its C-- output into LLVM rather than C code. Another project [Van Schei 2008] adds LLVM as an output option for the EHC compiler. However the Pure language—based on term-rewriting—targets LLVM specifically [Gräf 2010].

Agda and Cayenne have a relatively straightforward compilation strategy, outputting code in untyped functional languages. Edwin Brady's work on Epigram represents the most significant step in optimizing a dependently-typed language. He shows how to remove much of the runtime overhead of type-level computations and redundant annotations, as well as using an external representation of integers for speed. His contribution is substantial and goes beyond what's necessary for LambdaPi, which has neither lazy evaluation nor user-defined sum types.

1.5 Previous Work

This thesis builds on my experience with the compiler and interpreter for another simple dependently typed language. That project was performed in collaboration with Anders Mörtberg for David Sands' Frontiers of Programming Language Technology class, and thus we called it "foplang". The typechecker was based on Lennart Augustsson's response to [Löh, et al. 2001], where he presented an even simpler language.

The major difference with that work was that compilation targeted an abstract machine, similar to the SECD machine as described by Xavier LeRoy. The machine code was untyped and run using an interpreter written in Haskell. An interpreter existed in the compiler as well, in order to evaluate runtime types. Thus we avoided difficulties involved in interfacing to an existing code-generation API oriented towards conventional languages.

2 Implementation

The implementation work proceeded in three stages, in order to gain familiarity with the tools and techniques involved without being too overwhelming. First I built a simple calculator language I call LambdaMini, using the high-level Haskell bindings to LLVM. Then I had to use a lower-level interface to compile the simply-typed lambda calculus. And finally I integrated the parser and typechecker from [Löh et al., 2001] to add dependent types to the language. Counterintuitively, the last stage uses less information during code generation than does the second.

2.1 LambdaMini

The LambdaMini language contains only integer literals and arithmetic operators. There is a convenience mechanism for name-value bindings but no function abstraction or application. Each name-value binding is compiled to a low-level function, and all such functions have type `Int→Int`. As it turned out, this was an important restriction.

I implemented the LambdaMini compiler using the BNFC parser generator on the front-end, and Augustsson's high-level Haskell bindings for code generation. These bindings allow one to write natural-looking Haskell expressions that map directly to LLVM code. The trick is in using Haskell's type inference to determine and construct an explicit LLVM type at compile time. That is then transparently passed into the low-level LLVM calls to perform code generation.

The simplicity of LambdaMini, combined with the power of LLVM, means that all programs can be completely optimized at compile time! For instance this program:

```
def main = 3 * (42 + four);
def four = thirtythree - 29;
def thirtythree = 119 + 14 / 2 - 93;
```

becomes after optimization:

```
define i32 @main() {
  tail call void @printInt(i32 138)
  ret i32 0
}
```

Note the use of an external `printInt` function. This was defined in C and compiled separately. Ideally it would have been compiled to bitcode using llvm-gcc, allowing it to be inlined directly into the `main` function by LLVM's linker.

2.2 LambdaClassic

Aside from the learning curve of the LLVM bindings, LambdaMini was straightforward to implement. The next step was to add lambda calculus constructs. LambdaClassic adds booleans, comparison operators, conditional statements, and most importantly function abstraction and application. Only single-argument lambdas are permitted, but multi-argument lambdas could easily have been simulated, as they are for LambdaPi.

2.2.1 LLVM Bindings Rewrite

This extended language raised a couple major issues not present with LambdaMini. The first was a limitation of the high-level LLVM bindings. The conversion of Haskell types to LLVM types was done using the Haskell typeclass system. This works fine when writing programs with concrete types, which all have instances. There's also a recursive instance for function types, so LambaMini's single function type is okay.

But in LambdaClassic we do not know the type of every function and expression at compile time. The Haskell types of the expressions in the compiler are not concrete, and can't be mapped to LLVM types using only Haskell's type inference and typeclasses. One possible solution that was proposed by Norell, Swierstra and Augustsson was to rewrite the bindings to use GADTs for the mapping between Haskell and LLVM types. In the end, I decided to go the simple route and add additional high-level bindings that took an explicit type argument.

Originally I had expected to only replace bindings for declaring, defining and calling functions. However it turned out not to be feasible to just replace these without significant internal changes to the library. In the end I wrote my own bindings (the `CodeGen` and `Instruction` modules), building on a set of low-level bindings and utility functions that Augustsson and O'Sullivan had also provided. I made sure that my code generation monad could call itself recursively for handling nested lambdas, which the original library version could not do easily.

2.2.2 Free Variables and Closures

The second issue that arose with adding functions to the language was more fundamental. LLVM was designed to compile languages like C, with simple imperative functions. The natural approach would be to generate one LLVM function for each name-value binding in the language, plus a function for each lambda. But consider a function like:

```
add = \x : Int. \y : Int. x + y;
```

The innermost function takes `y` as its explicit argument, but also references the free variable `x`. So it's not enough to generate LLVM functions with one argument; we must also pass in any free variables referenced in the body of the lambda. A first approach might be to just add extra parameters to the functions we generate to pass along any free variables we need (lambda lifting). But consider this code:

```
map (add 3) xs
```

Here the add function is partially applied. We can't call it yet since we don't have all the arguments, but we need to remember the arguments we've been given far. This is exactly what closures are for; it just took me a while to realize they were unavoidable here.

The LambaClassic compiler has a pass to convert lambda nodes to closure nodes in the syntax tree. A closure node contains a list of all the free variable names and types in the lambda body, as determined by the `freeVars` function. The code generation phase handles closure nodes by allocating a closure object: a function pointer paired with a pointer to an environment, where the environment is itself a tuple. We can fill in the environment right away. Compiling a function application means taking the function pointer of the closure and passing it the environment in the first position and the its true argument in the second.

This is easier to show by example. Here's a simple multiply-add function:

```
multAdd = \x:Int. \y:Int. \z:Int. x * y + z
```

Here's what gets generated—in C-like pseudocode rather than real LLVM assembly for readability:

```
int multAdd() {
    closure { f = multAdd1, env = {} }
}
int multAdd1(environment, x) {
    closure { f = multAdd2, env = { x } }
}
int multAdd2(environment, y) {
    closure { f = multAdd3, env = { environment.x, y} }
}
int multAdd3(environment, z) {
    environment.x + environment.y + z
}
```

And here's what a call to `multAdd` looks like:

```
madd     = multAdd
partial1 = madd.f(madd1.env, x)
partial2 = partial1.f(partial1.env, y)
result   = partial2.f(partial2.env, z)
```

Each closure and environment structure is created using dynamic memory allocation. Thus there's a need for a garbage collector even for such a simple language. (The current implementation just leaks the memory.)

2.2.3 Compilation stages

1. Parse to BNFC-generated abstract syntax tree
2. Preprocess to internal AST representation
3. Convert lambda nodes to closure nodes
4. Code generation
5. Write code to bitcode file and link with main function written in C

The program's `main` function must have type `Int`. The main function written in C just calls the program's `main` function and prints the result.

2.3 LambdaPi

In moving from simply-typed lambda calculus to full LambdaPi we're adding a couple of things. One of course is the more sophisticated type checker, and the other is the runtime support for `Nat-Elim`, `Vec-Elim`, `Cons`, etc. The plan I followed was to replace my parser and typechecker with those provided by the LambdaPi authors, write the runtime as external C functions, and reuse my existing code generation as much as possible.

The parser and typechecker in the LambdaPi source code were relatively easy to extract. The parser converts to DeBruijn notation for easy of interpretation, whereas I found it easier to use named variables in my own code. So I must add in temporary variable names during preprocessing. Ideally the typechecker would return a syntax tree annotated with with type information, but its design makes this hard to do.

Writing the external constructor and eliminator functions was quite straightforward. All values in the runtime are boxed, which is to say they are treated as opaque pointers. There are no case statements in LambdaPi, so only arithmetic primitives and external functions need care how data is represented. `Nat`s are pointers to machine integers and `Vec`s are pointers to singly linked lists. Some of the external functions take closures as arguments, so I provide them the definition of a closure as a C struct.

External functions may be partially applied, and C functions of course cannot handle this. Thus we wrap calls to external functions in a series of lambdas. A call to `natElim` becomes:

```
\ _ mz mr n -> natElim(mz,mr,n)
```

The first argument is thrown away because it is a type argument that's of no use at run time.

The trickiest part of handling dependent types was how much type information to retain in the internal representation. Storing more type information would enable optimizations like partial unboxing. But due to the difficulties with creating a type annotated syntax tree I decided to compile to an untyped representation, as Agda and Cayenne do. Since LLVM is typed, this requires using an opaque pointer type for everything and inserting bitcasts where needed. Unfortunately this practice triggers a bug in LLVM 2.6 which prevented me from fully exploiting LLVM's optimizer.

2.3.1 Compilation stages

1. Parse to representation presented in the LambdaPi paper
2. Preprocess to internal, untyped AST representation, replacing integers and integer arithmetic with primitives, and constructor and eliminator calls with references to external primitives.
3. Convert lambda nodes to closure nodes
4. Code generation
5. Write code to bitcode file and link with main function written in C

2.3.2 Limitations

As previously noted, a proper implementation of a language with closures would require a garbage collector. Implementing one was considered outside the scope of this thesis, but LLVM does provide support for interfacing with a GC. Natural numbers are encoded as 32-bit integers. This could trivially be extended to 64-bit, but the proper solution would be to use a library like GMP for encoding arbitrary-size integers.

3 Optimization

There are a number of possible approaches for optimizing a dependently-typed language. Most have not been implemented in the current work either for lack of time or because of the simplicity of LambdaPi. Some of these optimizations are specific to dependently-typed languages, some are typical of functional languages in general, and some are low-level optimizations performed by LLVM itself.

3.1 Dependently-Typed Language Optimizations

In his thesis on Epigram, Edwin Brady considers a number of optimizations that reduce the overhead of dependent types while preserving their semantic guarantees. One strategy, impossible case removal, arguably allows for improvements beyond what could easily be achieved by an optimizer for a conventional language.

However, Epigram is a much more complex language than LambdaPi. The most significant difference is that LambdaPi lacks user-defined types, constructors and case statements. Most of the optimizations discussed by Brady are for efficient evaluation of case statements on user-defined types, and a reduction of the runtime information that need be stored or checked about them. One exception in his thesis is natural numbers. These are so important that Brady chooses to use an efficient external representation rather than defining them in the language. In the case of LambdaPi I define *all* primitive datatypes and operations externally.

Another difference is that LambdaPi is compiled to imperative machine code, while Epigram is run on the lazy, abstract G-machine. This means that Epigram's compiler need only do lambda-lifting and not concern itself with the details of closure generation. Finally, LambaPi uses built-in eliminators rather than recursion, so tail recursion is irrelevant.

Even without user-defined types we still will have type arguments and even entire functions that compute only with types. We may want to use type information for optimization purposes, but there's absolutely no reason to have type-level terms present at run time. The type-deletion strategies that Brady gives are complex, but those used by Cayenne are much simpler (see [Augustsson 1998], page 17). In short, the compiler calculates the type of each function parameter and each argument given in an application. If the term's type is not in the set of first-order types, the term is deleted.

Anders and I used a similar strategy for our foplang compiler, taking advantage of the fact that foplang annotated function arguments with their type. This is not the case with LambdaPi, and the design of its typechecker makes it difficult to write passes that annotate or modify the syntax tree returned by the parser. Type deletion would not be difficult, but it would require a rethinking of the typechecking algorithm.

One simple optimization that both I and Brady perform is to use primitive operations for arithmetic, rather than defining it within the language. He maps the function names like `plus` and `mult` to primitive operations he has added to the G-machine. With the appropriate flag my compiler will generate the corresponding LLVM instructions for these two functions. Like external operations, they are wrapped in lambdas to allow for partial application. Subtraction and division could easily be added. The typechecker still requires that LambdaPi definitions of these functions be in scope, since they could appear inside type annotations.

Dependently-typed languages typically use boxed values: values hidden behind a pointer so that they can be treated uniformly. It would be nice to specialize certain common cases, and Brady makes reference to a scheme where there is both a polymorphic function and a set of specializations, and the implementation is chosen via dynamic dispatch. But this requires keeping type information around at runtime, and he leaves this for future work.

3.2 Functional Language Optimizations

One challenge with compiling functional languages is how to handle partial application and over-application. LeRoy [2005] covers a number of strategies, and the one used here would be considered to be trivially in the eval-apply category, as it treats all functions as having a single argument. This was true of the original LambdaPi interpreter, as multi-argument lambdas are de-sugared to nested single-argument lambdas by the parser. This simplifies the implementation but generates excess code and unnecessary function call overhead.

Once we are dealing with multi-argument functions, we can use a more sophisticated eval-apply approach recommended by LeRoy and used in the OCaml native code compiler. All arity-r functions are compiled to take an environment plus r arguments. Applications are compiled to a call to the $\texttt{apply}_n$ operator. If $n = r$ then we can call the function normally. When $n < r$ we've got partial application, so we use a `curry` operator to wrap the function in r lambdas before applying it to its arguments (like I already do for external functions). For the over-application case of $n > r$ we perform the call with the first r arguments, then recursively `apply` the rest of them to the result.

The `apply` operator does its work at run time, since in general the arity of a function may not be known in the presence of higher-order functions. But in most cases we *can* determine a function's arity statically, especially with more sophisticated control flow analysis. In these cases we can just generate static code to handle over- or under-application. And in practice we usually will get the number of arguments we expect, and so we'll just be making a native function call.

3.3 Low-level optimizations

Brady notes in his thesis that traditional low-level compiler optimizations are still applicable to a language like Epigram, and that the language-specific passes he proposes in fact make it easier to apply them, since they remove indirection from the output code.

LLVM applies a large number of common optimization algorithms written for code in single static assignment form. These include inlining, constant folding, and constant subexpression elimination. The instruction combining pass in the latest release version has a bug when dealing my compiler's output, but otherwise it works well. Ideally the external functions would be compiled to LLVM bitcode using llvm-gcc. LLVM's linker performs whole-program optimizations, which would allow external functions to be inlined into calling code, and the result further optimized.

4 Conclusion and Future Work

In this thesis I have presented a compiler that translates a dependently-typed core calculus directly into an analogue of imperative assembly code. The large semantic gap between the two makes this a difficult task, and an even harder one to do efficiently. I approached the problem by writing three compilers for successively more complex languages, culminating in LambdaPi.

The LLVM Haskell bindings of O'Sullivan and Augustsson, while very clever and elegant for very simple compilers or specific code, are not up to this job. It could be worth using GADTs to infer LLVM types from more general Haskell types, and perhaps provide the same elegant interface for serious compilers. For the present work I have used lower-level bindings and a basic closure scheme for function calls.

To produce a production-quality compiler would require additional work. For one, it would need a garbage collector. LLVM currently provides support for GC, but considers the implementation to be out of its scope (though a simple, experimental collector is available). The current compiler passes around dummy values representing types and needs a type-deletion phase. Finally, some level of arity-based analysis would eliminate much of the functional call overhead and expose more optimization opportunities for LLVM.

I wish I had some more serious programs available in LambdaPi beyond what can be trivially built with the Prelude. It would be very interesting to compare equivalent programs run in the LambdaPi interpreter, my compiler, and an equivalent program in Agda. This is possible to a degree. Unfortunately the project is limited by the simplicity of LambdaPi. Without user-defined sum types, it's impossible to fully explore the sort of optimizations that Edwin Brady has pioneered with Epigram.

Acknowledgements

The concept for this thesis was suggested by Wouter Swierstra. The author wishes to thank Wouter, Ulf Norell, and Lennart Augustsson for their suggestions during the coding process, and Anders Mörtberg for his comments on the draft of this report.

Source Code

The source code for the compilers discussed here is available in a Darcs repository hosted on the Haskell community server.

`http://code.haskell.org/LambdaPiC`

References

AUGUSTSSON, L. 1998. Cayenne - a language with dependent types. In International Conference on Functional Programming, pages 239–250.

BRADY, E. 2005. Practical Implementation of a Dependently Typed Functional Programming Language. PhD Thesis, Computer Science Dept., University of Durham, England.

GRÄF, A. 2010. The Pure Programming Language. `http://pure-lang.googlecode.com/svn/docs/pure-intro/`

LATTNER, C. November 2008. Introduction to the LLVM Compiler System. ACAT 2008: Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy.

LEROY, X. 2005. From Krivine's machine to the Caml implementations, invited lecture, KAZAM workshop.

LÖH, A., MCBRIDE, C., and SWIERSTRA W. 2001. A tutorial implementation of a dependently typed lambda calculus. Fundamenta Informaticae XXI 1001–1031.

NORELL, U. 2007. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology.

OURY, N. and SWIERSTRA, W. 2008. The Power of Pi. The International Conference on Functional Programing, Vancouver, Canada.

TEREI, D. October 2009. Low Level Virtual Machine for Glasgow Haskell Compiler. Bachelor's Thesis, Computer Science and Engineering Dept., The University of New South Wales, Sydney, Australia.

VAN SCHEI, J. June 2008. Compiling Haskell to LLVM. Thesis Defense, Utrecht University, Netherlands.

XI, H. and PFENNING, F. June 1998. Eliminating array bound checking through dependent types. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 249–257, Montreal.