JFP **33**, e11, 9 pages, 2023.

doi:10.1017/S0956796823000084

FUNCTIONAL PEARL

A correct-by-construction conversion from lambda calculus to combinatory logic

WOUTER SWIERSTRA
Utrecht University, Utrecht, Netherlands
(*e-mail:* w.s.swierstra@uu.nl)

Abstract

This pearl defines a translation from well-typed lambda terms to combinatory logic, where both the preservation of types and the correctness of the translation are enforced statically.

1 Introduction

The correspondence between Curry's type-free lambda calculus and Schönfinkel's combinatory algebras is among the oldest known and the most aesthetically pleasing facts about the lambda calculus.
Peter Selinger, *The lambda calculus is algebraic, Journal of Functional Programming, 12(6), 549–566.*

This paper explores the connection between the lambda calculus and combinatory logic (Schönfinkel, 1924; Curry *et al.*, 1958). The terms of the lambda calculus are defined by the following grammar:

$$M \;::=\; x \mid M\,M \mid \lambda x.M$$

Evaluating and manipulating lambda terms require a careful treatment of variable binding. Combinatory logic, on the other hand, is a language without variable binding:

$$T \;::=\; x \mid T\,T \mid \mathsf{S} \mid \mathsf{K} \mid \mathsf{I}$$

Here, lambda abstractions have been replaced by three *combinators*: S, K, and I. Each combinator has its own reduction behaviour, given by the following rewrite rules:

$$\begin{aligned} \mathsf{S}\,f\,g\,x &\to (f\,x)\,(g\,x) \\ \mathsf{K}\,x\,y &\to x \\ \mathsf{I}\,x &\to x \end{aligned}$$

It is not so hard to define a translation from combinatory logic to lambda terms that preserves reduction behaviour. The following three lambda terms correspond to the combinators S, K, and I, respectively.

$$\lambda f\, g\, x\, .\, (f\, x)\, (g\, x)$$
$$\lambda x\, y\, .\, x$$
$$\lambda x\, .\, x$$

Interestingly, there is also a translation in the other direction, from lambda terms to combinatory logic. The key ingredient in this translation scheme is known as *bracket abstraction* or *combinatory abstraction*. Given a variable x and term in combinatory logic t, we can define the term $\Lambda\, x\, .\, t$ by means of the following three cases:

$$\begin{aligned} \Lambda\, x\, .\, x &= \mathsf{I} \\ \Lambda\, x\, .\, t &= \mathsf{K}\; t \quad \text{if } x \text{ does not occur freely in } t \\ \Lambda\, x\, .\, (t_1\; t_2) &= \mathsf{S}\; (\Lambda\, x\, .\, t_1)\, (\Lambda\, x\, .\, t_2) \end{aligned}$$

As its name suggests, the term in combinatory logic computed in this fashion simulates the reduction behaviour of a lambda abstraction in combinatory logic.

These translations are typically defined on untyped lambda terms. In this pearl, we try a different tack and explore how to prove that the translation from the simply typed lambda calculus to combinatory logic preserves both types and semantics. This is not a new result, but rather than prove these properties post hoc, we ensure the translation is correct by construction using the dependently typed programming language Agda (Norell, 2007).

2 Lambda calculus

To set the scene, we start by defining an evaluator for the simply typed lambda calculus. This evaluator features in numerous papers and introductions on programming with dependent types (McBride, 2004; Norell, 2009, 2013; Abel, 2016), yet we include it here in its entirety for the sake of completeness.

Types

The *types* of our lambda calculus consist of a single base type (ι) and functions between types, denoted using the function space operator ($\Rightarrow$):

```
data U : Set where
  ι   : U
  _⇒_ : U → U → U
```

We can map these types to their Agda counterparts.

```
Val : U → Set
Val ι          = A
Val (u₁ ⇒ u₂) = Val u₁ → Val u₂
```

https://doi.org/10.1017/S0956796823000084 Published online by Cambridge University Press

Here the interpretation of the base type, ι, is mapped to some type $\mathsf{A} : \mathsf{Set}$, which we pass as a parameter to this development; the functions and proofs that follow do not depend on the interpretation of our base type in any meaningful way.

Before defining lambda terms, we need one last definition. We will represent *contexts* or *type environments* as lists of types:

$$\mathsf{Ctx} = \mathsf{List}\,\mathsf{U}$$

Typically, we will use variable names drawn from the Greek alphabet to refer to types (such as σ and τ) and contexts (Γ and Δ).

Terms

Before we define the *terms* of the simply typed lambda calculus, we need to decide on how to treat variables. We begin by defining the following inductive family, modelling valid references to a type σ in a given context Γ:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Ref}\,(\sigma : \mathsf{U}) : \mathsf{Ctx} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{zero} : \mathsf{Ref}\,\sigma\,(\sigma :: \Gamma) \\
\quad \mathsf{succ} : \mathsf{Ref}\,\sigma\,\Gamma \to \mathsf{Ref}\,\sigma\,(\tau :: \Gamma)
\end{array}
$$

Erasing the type indices, we are left with the Peano natural numbers – corresponding to the typical De Bruijn representation of variable binding.

We can now define the datatype for well-typed, well-scoped lambda terms as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Term} : \mathsf{Ctx} \to \mathsf{U} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{app} : \mathsf{Term}\,\Gamma\,(\sigma \Rightarrow \tau) \to \mathsf{Term}\,\Gamma\,\sigma \to \mathsf{Term}\,\Gamma\,\tau \\
\quad \mathsf{lam} : \mathsf{Term}\,(\sigma :: \Gamma)\,\tau \to \mathsf{Term}\,\Gamma\,(\sigma \Rightarrow \tau) \\
\quad \mathsf{var} : \mathsf{Ref}\,\sigma\,\Gamma \to \mathsf{Term}\,\Gamma\,\sigma
\end{array}
$$

Each constructor mirrors a familiar typing rule: applications require the function's domain and argument's type to coincide; lambda abstractions introduce a new variable in the context of the lambda's body; the var constructor may be used to refer to any variable that is currently in scope.

Evaluation

The dependent types in the definition of Term pay dividends once we try to define an evaluator for lambda terms. Before we can do so, however, we need to introduce a datatype for *environments*:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Env} : \mathsf{Ctx} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{nil} \quad : \mathsf{Env}\,[] \\
\quad \mathsf{cons} : \mathsf{Val}\,\sigma \to \mathsf{Env}\,\Gamma \to \mathsf{Env}\,(\sigma :: \Gamma)
\end{array}
$$

An environment stores a value for each variable in the context Γ, as witnessed by the following lookup function:

$$
\begin{array}{l}
\mathsf{lookup} : \mathsf{Ref}\,\sigma\,\Gamma \to \mathsf{Env}\,\Gamma \to \mathsf{Val}\,\sigma \\
\mathsf{lookup}\,\mathsf{zero} \qquad (\mathsf{cons}\,\mathsf{x}\,\mathsf{env}) = \mathsf{x} \\
\mathsf{lookup}\,(\mathsf{succ}\,\mathsf{i})\,(\mathsf{cons}\,\mathsf{x}\,\mathsf{env}) = \mathsf{lookup}\,\mathsf{i}\,\mathsf{env}
\end{array}
$$

https://doi.org/10.1017/S0956796823000084 Published online by Cambridge University Press

Note that this function is *total*. The type indices ensure that there is no valid variable in the empty context; correspondingly, the lookup function need never worry about returning a value when the environment is empty.

We can now define an evaluator for the simply typed lambda calculus:

```
[[_]] : Term Γ σ → (Env Γ → Val σ)
[[ app t₁ t₂ ]] = λ env → ([[ t₁ ]] env) ([[ t₂ ]] env)
[[ lam t ]]     = λ env → λ x → [[ t ]] (cons x env)
[[ var i ]]     = λ env → lookup i env
```

That this code type checks at all is somewhat surprising at first. It maps app constructors to Agda's application and lam constructors to Agda's built-in lambda construct. Once again, the type indices ensure that the evaluation of the lam construct must return a function (and hence we may introduce a lambda). Similarly in the case for applications, evaluating t_1 will return a function whose domain coincides with the type of the value arising from the evaluation of t_2. Finally, the environment of type Env Γ passed as an argument contains just the right values for all the variables drawn from the context Γ.

3 Translation to combinatory logic

Before we can define the translation from lambda terms to combinators, we need to fix our target language. As a first attempt, we might try something along the following lines, turning the grammar from the introduction into an Agda datatype:

```
data Comb : Set where
  S K I : Comb
  app : Comb → Comb → Comb
  var : ... → Comb
```

Yet if we aim for our translation to be type-preserving, the very least we can do is decorate our combinators with the same type information as our lambda terms:

```
data Comb (Γ : Ctx) : U → Set where
  S   : Comb Γ ((σ ⇒ τ ⇒ τ') ⇒ (σ ⇒ τ) ⇒ (σ ⇒ τ'))
  K   : Comb Γ (σ ⇒ τ ⇒ σ)
  I   : Comb Γ (σ ⇒ σ)
  app : Comb Γ (σ ⇒ τ) → Comb Γ σ → Comb Γ τ
  var : Ref σ Γ → Comb Γ σ
```

The types of both the app and var constructors are the same as we saw for the lambda terms. The types of the primitive combinators are determined by their desired reduction behaviour. Note that – as our Comb lacks lambdas and cannot introduce new variables – the context is now a *parameter* rather than an *index* as we saw for the Term datatype. This is the essence of combinatory logic: a language with variables but without binders.

Yet we will strive to do even better. We will define a translation that preserves both the types and *dynamic semantics* of our lambda terms. To achieve this, we index our combinators with *both* their types and their intended semantics, given by a function of type

$\mathsf{Env}\,\Gamma \rightarrow \mathsf{Val}\,\mathsf{u}$. This will enable us to define a translation from a lambda term to a term in combinatory logic that has the same semantics as its input lambda term. This yields the final version of our datatype for combinatory logic:

```
data Comb : (Γ : Ctx) → (u : U) → (Env Γ → Val u) → Set where
  S   : Comb Γ ((σ ⇒ τ ⇒ τ') ⇒ (σ ⇒ τ) ⇒ σ ⇒ τ') (λ env → λ f g x → (f x) (g x))
  K   : Comb Γ (σ ⇒ (τ ⇒ σ)) (λ env → λ x y → x)
  I   : Comb Γ (σ ⇒ σ) (λ env → λ x → x)
  var : (i : Ref σ Γ) → Comb Γ σ (λ env → lookup i env)
  app : Comb Γ (σ ⇒ τ) f → Comb Γ σ x → Comb Γ τ (λ env → (f env) (x env))
```

Here the type of each base combinator (S, K, and I) contains both its type and semantics. For example, the I combinator has type $\sigma \Rightarrow \sigma$ and corresponds to the lambda term $\lambda\,x \rightarrow x$. None of the combinators rely on the additional environment parameter env. This environment is used in the var constructor; just as we saw in our evaluator for lambda terms, this environment stores a value for each variable. Finally, the app constructor applies one combinator term to another. The type information for both the var and app constructors coincides with their counterparts from the Term data type; their intended semantics can be read off from the evaluator for lambda terms, $[\![\,t\,]\!]$, that we defined previously.

The key difference between lambda terms and SKI combinators is the lack of lambdas in the latter. To handle the *bracket abstraction* translation from the introduction, we define the abs function that maps one combinator term to another:

```
abs : ∀ {f} → Comb (σ :: Γ) τ f → Comb Γ (σ ⇒ τ) (λ env x → f (cons x env))
abs S               = app K S
abs K               = app K K
abs I               = app K I
abs (app t₁ t₂)     = app (app S (abs t₁)) (abs t₂)
abs (var zero)      = I
abs (var (succ i))  = app K (var i)
```

This behaviour of the abs function should be clear from its type: given a Comb term of type τ using variables drawn from the context $\sigma :: \Gamma$, the abs function returns a combinator of type $\sigma \Rightarrow \tau$ using variables drawn from the context Γ. Essentially, any occurrences of the var Top are replaced with the identity I; the new argument is distributed over applications using the S combinator; any other variables or base combinators discard this new argument by introducing an additional K combinator.

With this definition in place, we can now define our type-preserving correct-by-construction translation. That is, we aim to define a translation with the following type:

```
translate : (t : Term Γ σ) → Comb Γ σ ⟦ t ⟧
```

Here a lambda term of type σ in the context Γ is mapped to a combinator of type σ using variables drawn from the context Γ in such a way that the evaluation of t and semantics of the combinator are identical, namely $[\![\,t\,]\!]$. The definition of this translation is now entirely straightforward.

https://doi.org/10.1017/S0956796823000084 Published online by Cambridge University Press

$$
\begin{aligned}
\mathsf{translate}\,(\mathsf{app}\,t_1\,t_2) &= \mathsf{app}\,(\mathsf{translate}\,t_1)\,(\mathsf{translate}\,t_2)\\
\mathsf{translate}\,(\mathsf{lam}\,\mathsf{t}) &= \mathsf{abs}\,(\mathsf{translate}\,\mathsf{t})\\
\mathsf{translate}\,(\mathsf{var}\,\mathsf{i}) &= \mathsf{var}\,\mathsf{i}
\end{aligned}
$$

To see why this code type checks, note that both the (dynamic) semantics of both the app and var constructors of the Comb datatype coincide precisely with their semantics as lambda terms, $[\![\,\mathsf{app}\,t_1\,t_2\,]\!]$ and $[\![\,\mathsf{var}\,\mathsf{i}\,]\!]$, respectively. Finally, if translating the body of a lambda produces some Comb term f, the abs function produces a combinator term with the semantics $\lambda\,\mathsf{env}\,\mathsf{x} \rightarrow \mathsf{f}\,(\mathsf{Cons}\,\mathsf{x}\,\mathsf{env})$. The similarity between the *type* of the abs function and the lam branch of our evaluator is no coincidence.

There is a subtle difference between this translation scheme and the one presented in the introduction. In particular, when a variable does not occur anywhere, the bracket abstraction sketched in the introduction immediately introduces a K combinator, whereas the abs function will use the S combinator in every application – even if the variable is unused in both branches. This may lead to unnecessarily large combinatorial terms. Furthermore, the SKI-combinators are not the only possible choice of combinatorial basis. In particular, the S combinator *always* passes its third argument to the first two – even if it is unused in one of the branches. Can we do better?

4 An optimising translation

There is an alternative implementation of bracket abstraction, using two additional combinators B and C, that Turner (1979) attributes to Curry. The reduction behaviour of B and C is defined as follows:

$$
\begin{aligned}
\mathsf{B}\,f\,g\,x &\rightarrow f\,(g\,x)\\
\mathsf{C}\,f\,g\,x &\rightarrow (f\,x)\,g
\end{aligned}
$$

In contrast to the S combinator, the B combinator only passes its third argument to its second argument. The C combinator, on the other hand, only passes its third argument to its first argument. This avoids unnecessarily duplicating the third argument x, when it is only used by one of the two terms in an application. When the variable is not used at all, we can introduce the K combinator as suggested by the translation scheme from the introduction. As a result, normalising terms may require fewer reduction steps.

We can readily extend our Comb datatype with new constructors for these two combinators:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Comb} : (\Gamma : \mathsf{Ctx}) \rightarrow (\mathsf{u} : \mathsf{U}) \rightarrow (\mathsf{Env}\,\Gamma \rightarrow \mathsf{Val}\,\mathsf{u}) \rightarrow \mathsf{Set}\ \textbf{where}\\
&\quad \ldots\\
&\quad \mathsf{B} : \mathsf{Comb}\,\Gamma\,((\sigma \Rightarrow \tau) \Rightarrow (\rho \Rightarrow \sigma) \Rightarrow (\rho \Rightarrow \tau))\,(\lambda\,\mathsf{env}\,\mathsf{f}\,\mathsf{g}\,\mathsf{x} \rightarrow \mathsf{f}\,(\mathsf{g}\,\mathsf{x}))\\
&\quad \mathsf{C} : \mathsf{Comb}\,\Gamma\,((\sigma \Rightarrow \tau \Rightarrow \rho) \Rightarrow \tau \Rightarrow \sigma \Rightarrow \rho)\,(\lambda\,\mathsf{env}\,\mathsf{f}\,\mathsf{g}\,\mathsf{x} \rightarrow (\mathsf{f}\,\mathsf{x})\,\mathsf{g})
\end{aligned}
$$

When translating an application, we now need to select between four possible choices: K, B, C, and S, depending how variables are used. How can we make this choice, while still guaranteeing that types and semantics are preserved accordingly?

The key insight is that the translation scheme, implemented by the abs function above, already informs us whether or not a variable is used: any variable occurrence or combinator

https://doi.org/10.1017/S0956796823000084 Published online by Cambridge University Press

that does not use the most recently bound variable starts with an application of the K combinator. Rather than indiscriminately apply the S combinator on subterms, we can instead differentiate where variables are actually used. To this end, we define the following specialised function for applying the S combinator:

$$\begin{array}{l}
\mathsf{sapp} : \forall \{\mathsf{f\,x}\} \rightarrow \mathsf{Comb}\,\Gamma\,(\sigma \Rightarrow \tau \Rightarrow \rho)\,\mathsf{f} \rightarrow \mathsf{Comb}\,\Gamma\,(\sigma \Rightarrow \tau)\,\mathsf{x} \rightarrow \\
\quad \mathsf{Comb}\,\Gamma\,(\sigma \Rightarrow \rho)\,(\lambda\,\mathsf{env}\,\mathsf{y} \rightarrow (\mathsf{f}\,\mathsf{env}\,\mathsf{y})\,(\mathsf{x}\,\mathsf{env}\,\mathsf{y}))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{sapp}\,(\mathsf{app}\,\mathsf{K}\,t_1)\,\mathsf{I} & = & t_1 \\
\mathsf{sapp}\,(\mathsf{app}\,\mathsf{K}\,t_1)\,(\mathsf{app}\,\mathsf{K}\,t_2) & = & \mathsf{app}\,\mathsf{K}\,(\mathsf{app}\,t_1\,t_2) \\
\mathsf{sapp}\,(\mathsf{app}\,\mathsf{K}\,t_1)\,t_2 & = & \mathsf{app}\,(\mathsf{app}\,\mathsf{B}\,t_1)\,t_2 \\
\mathsf{sapp}\,t_1\,(\mathsf{app}\,\mathsf{K}\,t_2) & = & \mathsf{app}\,(\mathsf{app}\,\mathsf{C}\,t_1)\,t_2 \\
\mathsf{sapp}\,t_1\,t_2 & = & \mathsf{app}\,(\mathsf{app}\,\mathsf{S}\,t_1)\,t_2
\end{array}$$

Unlike the previous naive translation, this definition avoids unnecessary occurrences of the K combinator, simplifying the resulting definition whenever possible. Only the very last case, when neither t_1 nor t_2 start with an application of K, introduces the S combinator. The other cases introduce an outermost K, B, or C combinator, depending on where the 'bound' variable occurs.

To complete the translation, we need to adapt the abs function: adding new cases for B and C, and calling the sapp function instead of applying S directly.

$$\mathsf{abs} : \forall \{\mathsf{f}\} \rightarrow \mathsf{Comb}\,(\sigma :: \Gamma)\,\tau\,\mathsf{f} \rightarrow \mathsf{Comb}\,\Gamma\,(\sigma \Rightarrow \tau)\,(\lambda\,\mathsf{env}\,\mathsf{x} \rightarrow \mathsf{f}\,(\mathsf{cons}\,\mathsf{x}\,\mathsf{env}))$$

$$\begin{array}{lcl}
\ldots & & \\
\mathsf{abs}\,\mathsf{B} & = & \mathsf{app}\,\mathsf{K}\,\mathsf{B} \\
\mathsf{abs}\,\mathsf{C} & = & \mathsf{app}\,\mathsf{K}\,\mathsf{C} \\
\mathsf{abs}\,(\mathsf{app}\,t_1\,t_2) & = & \mathsf{sapp}\,(\mathsf{abs}\,t_1)\,(\mathsf{abs}\,t_2)
\end{array}$$

The types and remaining cases definitions, however, remain unchanged.

5 Reflection

Although the translation schemes are reasonably straightforward, finding the implementation presented here was not. Writing dependently typed programs in this style – folding a program's specification into its type – may feel like a bit of a parlour trick, where the right choice of definitions ensures the entire construction is correct. Yet reading through these definitions after the fact – like so often with Agda programs – does not tell the complete story of how they were constructed.

Verifying the type safe translation from lambda terms to SKI combinators is a question I have set my students in the past. Proving this translation correct requires defining an evaluation function for combinatory terms and then proving that the translation is semantics preserving. Interestingly, this proof requires an axiom – functional extensionality – in the case for lambdas, as we need to prove two functions equal. Yet the *structure* of proof is simple enough: it relies exclusively on induction hypotheses and a property of the abs function. It is this observation that makes it possible to incorporate the correctness proofs in the definitions themselves – where the required property of the abs function is combined with its definition. This observation is an instance of the *recomputation lemma* of algebraic

https://doi.org/10.1017/S0956796823000084 Published online by Cambridge University Press

ornaments (McBride, 2010). Extending the translation scheme to use the B and C combinators is a bit harder. The code accompanying this paper demonstrates how to use the 'co-De Bruijn' representation of variables to define the optimising translation (McBride, 2018). Ralf Hinze suggested defining the translation directly using the sapp function.

Historically, combinatory logic arose from the desire to find a foundation for mathematics that avoided the issues surrounding variable binding (Schönfinkel, 1924; Curry *et al.*, 1958). The translation between between lambda calculus and combinatory logic is well documented in numerous textbooks (see Barendregt, 1984, Chapter 7; Hindley & Seldin, 1986, Chapter 2; Sørensen & Urzyczyn, 2006, Chapter 5.4; Mimram, 2020, Chapter 3.6). There is a close connection between combinatory logic and Hilbert-style proof systems – cognoscenti will recognise the correspondence between the first three axiom schemes and the types that can be assigned to the three combinators above. Since then, Turner (1979) has explored how to compile functional programs to combinatory logic (see also Peyton Jones, 1987, Chapter 16; Diller, 1988). This idea has been extended further by Hughes (1982) and many others, even leading to design of custom hardware for efficiently rewriting terms in combinatory logic (Stoye, 1983, 1985; Scheevel, 1986). The lambda terms corresponding to the S and K combinators have made a recent reappearance as the operations defining the Reader applicative functor (McBride & Paterson, 2008).

As our starting point, we have taken the 'traditional' simply typed lambda calculus. More recent work by Kiselyov (2018) shows how a slight modification to the typing rules allows for a denotational semantics as combinators directly. Formalising this in a proof assistant, however, is left as an exercise for the reader.

Acknowledgments

I would like to thank Ralf Hinze for his encouragement to keep this pearl short. His suggestions helped to simplify Section 4 enormously. The anonymous JFP reviewers provided further constructive comments.

Conflicts of interest

None.

References

Abel, A. (2016) Agda tutorial. In 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4–6, 2016, Proceedings. Springer.

Barendregt, H. (1984) *The Lambda Calculus: Its Syntax and Semantics*. Elsevier.

Curry, H. B., Feys, R., Craig, W., Hindley, J. R. & Seldin, J. P. (1958) *Combinatory Logic*, vol. 1. North-Holland Amsterdam.

Diller, A. (1988) *Compiling Functional Languages*. John Wiley & Sons.

Hindley, J. R. & Seldin, J. P. (1986) *Introduction to Combinators and λ-Calculus*. Cambridge University Press.

Hughes, R. J. M. (1982) Super-combinators a new implementation method for applicative languages. In Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, pp. 1–10.

Kiselyov, O. (2018) λ to SKI, semantically. In International Symposium on Functional and Logic Programming Springer, pp. 33–50.

McBride, C. (2004) Epigram: Practical programming with dependent types. In International School on Advanced Functional Programming, vol. 3622. Springer, pp. 130–170.

McBride, C. (2010) Ornamental algebras, algebraic ornaments. *Submitted to J. Funct. Program*.

McBride, C. (2018) Everybody's got to be somewhere. *Electron. Proc. Theoret. Comput. Sci.* **275**, 53–69.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Mimram, S. (2020) *PROGRAM = PROOF*. http://program-proof.mimram.fr.

Norell, U. (2007) *Towards a Practical Programming Language based on Dependent Type Theory*. PhD Thesis, Chalmers University of Technology.

Norell, U. (2009) Dependently typed programming in Agda. In Advanced Functional Programming: 6th International School AFP. Springer Berlin Heidelberg, pp. 230–266.

Norell, U. (2013) Interactive programming with dependent types. In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP'13. ACM, pp. 1–2.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Scheevel, M. (1986) NORMA: A graph reduction processor. In Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 212–219.

Schönfinkel, M. (1924) Über die Bausteine der mathematischen Logik. *Mathematische Annalen* **92**(3), 305–316.

Sørensen, M. H. & Urzyczyn, P. (2006) *Lectures on the Curry-Howard Isomorphism*. Elsevier.

Stoye, W. (1983) *The SKIM Microprogrammer's Guide*. Technical Report UCAM-CL-TR-40. University of Cambridge, Computer Laboratory.

Stoye, W. R. (1985) *The Implementation of Functional Languages Using Custom Hardware*. PhD Thesis, University of Cambridge, Computer Laboratory.

Turner, D. A. (1979) Another algorithm for bracket abstraction. *J. Symb. Logic* **44**(2), 267–270.