FUNCTIONAL PEARL

A type-correct, stack-safe, provably correct expression compiler in EPIGRAM

JAMES MCKINNA

University of St. Andrews

(*e-mail:* `james.mckinna@st-andrews.ac.uk`)

JOEL WRIGHT

University of Nottingham

(*e-mail:* `jjw@cs.nott.ac.uk`)

Abstract

Conventional approaches to compiler correctness, type safety and type preservation have focused on off-line proofs, either on paper or formalised with a machine, of existing compilation schemes with respect to a reference operational semantics. This pearl shows how the use of *dependent types* in programming, illustrated here in EPIGRAM, allows us not only to build-in these properties, but to write programs which guarantee them by *design* and subsequent *construction*.

We focus here on a very simple expression language, compiled into tree-structured code for a simple stack machine. Our purpose is not to claim any sophistication in the source language being modelled, but to show off the metalanguage as a tool for writing programs for which the type preservation and progress theorems are self-evident by construction, and finally, whose correctness can be proved directly in the system.

In this simple setting we achieve the following;

- a type-preserving evaluation semantics, which takes *typed* expressions to *typed* values.
- a compiler, which takes *typed* expressions to *stack-safe* intermediate code.
- an interpreter for compiled code, which takes stack-safe intermediate code to a big-step stack transition.
- a compiler correctness proof, described via a function whose type expresses the equational correctness property.

---

1 Introduction

“This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language” (McCarthy & Painter, 1967). It is forty years since McCarthy pioneered the certification of programming language implementation: his approach emphasised abstract syntax, operational semantics, definition and proof by structural induction, and is largely unchanged to the present day, with correctness properties expressed via commuting diagrams of the form illlustrated below. What *has* changed is the emergence of systems for

checking proofs, and through the specific use of tools based on varieties of Martin-Löf type theory, the possibility of integrating programming and proof in one unified formalism.

$$\begin{tikzcd} Exp \arrow[r, "eval"] \arrow[d, "compile"'] & Value \arrow[d, "push"] \\ Code \arrow[r, "exec"'] & Stack \to Stack \end{tikzcd}$$

This paper examines a simple example, that of a typed language of arithmetic and boolean expressions with two semantics given by a primitive-recursive interpreter, **eval**, and a compiler, **compile**, to tree-structured code for a simple stack-based abstract machine with intepreter **exec**. It thus contributes indirectly to the POPLMARK challenge (Aydemir *et al.*, 2005) in illustrating a completely checkable and executable piece of programming language manipulation. It is a prototype for more substantial experiments which we intend to report upon in future work, although unlike McCarthy, we do *not* envisage that in order to make such extensions, "a complete revision of the formalism will be required" (*ibid.*, closing remark).

2 Dependently Typed Programming in Epigram

The use of a dependently-typed host language allows us a direct formulation of type preservation and progress. Dependent type theory provides programmers with more than just an integrated logic for reasoning about program correctness — it allows more precise types for programs and data in the first place, thus strengthening the typechecker's language of guarantees.

EPIGRAM (McBride & McKinna, 2004), a kernel language for dependently-typed programming, supports Dybjer's notion of *inductive families* (Dybjer, 1994) as its language of data, with an economical syntax for function (program) definition. The syntax of the source code presented in this paper is that originally presented in 'The view from the left' (2004), however in the interest of readability we suppress explicit calls to case constructs (which can be formally justified (Goguen *et al.*, 2006)). Type signature definitions are presented uniformly via a two-dimensional 'inference rule' style; this is used throughout to introduce new datatype families via the data keyword, their constructors via where, and new function symbols via let. Functions are declared by giving their type signatures, followed by a tree structure which superficially resembles the equational syntax for pattern matching programs in SML or Haskell.

Inductive families, as supported in EPIGRAM, allow us to represent *directly* the stratification of values and expressions by their types. The interested reader is referred to McBride & McKinna (2004) for further details on EPIGRAM, and for the program in this paper, to the fully annotated epigram source:

`http://www.e-pig.org/downloads/compiler_pearl-2006-07-19.epi`

3 The First Semantics : eval

The example of a well-typed interpreter is, of course, familiar as the illustration of programming with GADTs (for example in Hinze (Gibbons & de Moor, 2003)), but whose earliest appearance in the literature we can find is that of Augustsson and Carlsson in the dependently typed language Cayenne (1999), although such examples doubtless exist further back, at least in the folklore. GADTs are themselves a restricted form of type family, allowing non-uniform indexing over *host-language* types. One advantage of our approach via full inductive families is that we maintain a clean separation between the object-language type system and its model in the host language, where the use of GADTs relies on the pun between the two levels. Moreover, to extend the example beyond the simple evaluator, for example to embrace well-typed stacks as we do below, requires further exploitation of such tricks, in the style first identified (and argued against!) by McBride (2002). We can only imagine what contortions might be required to represent the correctness proof in Section 5 in such a style.

3.1 Type preservation is the type of the interpreter

The title of this section emphasises the basic feature of language representation available to the programmer working in the dependently-typed setting: namely that properties of programs (in this case the object-language semantics) become directly expressible via the type system. Here we achieve this by stratifying the representation of object-language expressions by their object-language types.

We begin by introducing this language of type expressions:

$$\underline{\text{data}} \quad \frac{}{\mathsf{TyExp} \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{nat} \ : \ \mathsf{TyExp}} \quad \frac{}{\mathsf{bool} \ : \ \mathsf{TyExp}}$$

Now, EPIGRAM supports the following definition of the host language family of types, $\mathbf{Val}\,T$, indexed by T : TyExp, of object-language values, by case analysis:

$$\underline{\text{let}} \quad \frac{T \ : \ \mathsf{TyExp}}{\mathbf{Val}\,T \ : \ \star} \qquad \begin{array}{l} \mathbf{Val}\,\mathsf{nat} \Rightarrow \mathsf{Nat} \\ \mathbf{Val}\,\mathsf{bool} \Rightarrow \mathsf{Bool} \end{array}$$

where Nat and Bool are the (usual) host-language inductive definitions of Peano naturals and Booleans (with true, false and **cond**), respectively, omitted here.

Finally the inductive family of expressions, indexed by T : TyExp, may be given directly as follows:

$$\underline{\text{data}} \quad \frac{T \ : \ \mathsf{TyExp}}{\mathsf{Exp}\,T \ : \ \star} \quad \underline{\text{where}}$$

$$\frac{v \ : \ \mathbf{Val}\,T}{\mathsf{val}\,v \ : \ \mathsf{Exp}\,T} \qquad \frac{e_1, e_2 \ : \ \mathsf{Exp}\,\mathsf{nat}}{\mathsf{plus}\,e_1\,e_2 \ : \ \mathsf{Exp}\,\mathsf{nat}} \qquad \frac{b \ : \ \mathsf{Exp}\,\mathsf{bool}\,; \ e_1, e_2 \ : \ \mathsf{Exp}\,T}{\mathsf{if}\,b\,e_1\,e_2 \ : \ \mathsf{Exp}\,T}$$

which declares the (types of the) constructors of the abstract syntax essentially in terms of their informal typing rules. Notice that we get object-level polymorphism in the host-level injection of values into expressions.

Throughout, we have made extensive use of EPIGRAM's *implicit syntax* mechanisms (adapted from Pollack's (Pollack, 1990) original approach in LEGO) to suppress the object-level indices T. In particular, the if constructor is polymorphic at the object level via the dependency at the host level, and any instance will have its type indices correctly inferred.

Now we are in a position to write the evaluation function **eval**, taking expressions to values. But now we have explicit (object-level type) index information in the (host-level) types, we can express type preservation *directly* in the type signature of **eval**:

$$\underline{\mathrm{let}} \quad \frac{e \;:\; \mathsf{Exp}\, T}{\mathbf{eval}\, e \;:\; \mathbf{Val}\, T}$$

The object langauge *property* of type preservation has been reified as a host-language *type*. Constructing a program body for the interpreter with the above signature is now, as usual, by structural recursion on $e \;:\; \mathsf{Exp}\, T$

$$\begin{array}{lll} \mathbf{eval}\, e \Leftarrow \underline{\mathsf{rec}}\, e & & \\ \quad \mathbf{eval}\, (\mathsf{val}\, v) & \Rightarrow & v \end{array}$$

$$\begin{array}{lcr} \mathbf{eval}\,(\mathsf{plus}\,e_1\,e_2) & \Rightarrow & (\mathbf{eval}\,e_1) + (\mathbf{eval}\,e_2) \\ \mathbf{eval}\,(\mathsf{if}\,b\,e_1\,e_2) & \Rightarrow & \mathbf{cond}\,(\mathbf{eval}\,b)\,(\mathbf{eval}\,e_1)\,(\mathbf{eval}\,e_2) \end{array}$$

where + is host-language addition on Nat, and **cond** is host-language if-then-else in the usual way. EPIGRAM's typechecker enforces the object-language typing rules we have encoded in the definition of $\mathsf{Exp}\,T$, which ensure, inductively, that recursive calls on **eval** yield values of the correct object- *and* host-level types.

Not only have we expressed our desired preservation property as a type, but the proof that it holds is expressed precisely by the program for **eval** itself! That is, for the recursive definition of **eval** to have the host level type claimed, is precisely the proof that **eval** satisfies object-level type preservation. By working in a rich host language, we obtain an extremely terse, type-correct interpreter virtually for free.

4 The Second Semantics : compile & exec

For the purposes of this paper we consider a direct-style semantics obtained by compiling to code for a simple stack-based abstract machine. In this setting there is a clear, well-defined, notion of safety, namely:

stack-type safety stacks are typed; intermediate code is stack-type respecting, in a way to be made clear below; in particular, code for addition expects to pop two natural number arguments on the stack, and push back a single natural number;

no underflow intermediate code executes only in the context of a stack which has enough of the right type of arguments at the top to continue execution.

EPIGRAM's type system allows us to represent both of these properties (preservation and progress, again) in the types of data (typed stacks; typed intermediate code) and operations (compile; exec) respectively, so that *no further work* is required to establish them. This is simply another instance of the idea that "type preservation is the type of the interpreter". The corresponding progress theorem is encoded in the types of intermediate object-level code fragments.

4.1 Typing stacks

Our simple abstract machine will be defined in terms of a big-step semantics for intermediate code *Code*, taking an initial stack of *Val*s to a result stack of *Val*s. We exploit the same idea as before, namely to index the family of stacks over their object-level type signature: these stack types may be given simply as lists of *TyExp*s, where lists are declared in the usual way with nil ([]) and cons (::):

$$\underline{\text{data}} \quad \frac{A \;:\; \star}{\mathsf{List}\ A \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{[] \;:\; \mathsf{List}\ A} \quad \frac{x \;:\; A \qquad xs \;:\; \mathsf{List}\ A}{x :: xs \;:\; \mathsf{List}\ A}$$

$$\underline{\text{let}} \quad \frac{}{\mathbf{StackType} \ : \ \star} \qquad \mathbf{StackType} \Rightarrow \mathsf{List\ TyExp}$$

Typed stacks are now the family of dependently-typed lists, indexed by stack type:

$$\underline{\text{data}} \quad \frac{S \ : \ \mathbf{StackType}}{\mathsf{Stack}\, S \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\varepsilon \ : \ \mathsf{Stack}\, []} \quad \frac{v \ : \ \mathbf{Val}\, T \qquad s \ : \ \mathsf{Stack} S}{v \rhd s \ : \ \mathsf{Stack}(T :: S)}$$

The use of dependent types suppports such an entirely concrete approach to stacks: since we enforce stack-typing, we can specify a type-safe **top** operation without needing to handle stack underflow explictly, with the obvious definition:

$$\underline{\text{let}} \quad \frac{s \ : \ \mathsf{Stack}(T :: S)}{\mathbf{top}\, s \ : \ \mathbf{Val}\, T} \qquad \mathbf{top}\,(v \rhd s) \Rightarrow v$$

Epigram accepts such case analysis during type-checking, correctly rejecting the need to consider the ε case, since its type fails to unify with $\mathsf{Stack}(T :: S)$. A detailed account of typechecking such pattern matching programs is available elsewhere (Goguen *et al.*, 2006); the precise details need not concern us here.

4.2 Compiling and executing typed intermediate code

Stack-safety for intermdiate code is achieved in two steps: firstly, we *define* the type family Code of intermediate code in such a way that the type of its interpreter **exec** expresses the stack-type preservation theorem. Then we *define* the compiler **compile** to produce code with the intended meaning, namely to leave a value of the correct type on top of the stack:

$$\underline{\mathrm{data}} \quad \frac{S, S' \ : \ \mathbf{StackType}}{\mathsf{Code}\, S\, S' \ : \ \star} \quad \cdots$$

$$\underline{\mathrm{let}} \quad \frac{c \ : \ \mathsf{Code}\, S\, S' \ ; \ s \ : \ \mathsf{Stack}\, S}{\mathbf{exec}\, c\, s \ : \ \mathsf{Stack}\, S'} \quad \cdots$$

$$\underline{\mathrm{let}} \quad \frac{e \ : \ \mathsf{Exp}\, T}{\mathbf{compile}\, e \ : \ \mathsf{Code}\, S\, (T :: S)} \quad \cdots$$

4.3 Specifying Intermediate Code

For our specific expression language, we can now introduce actual intermediate code as (tree-structured) sequences, with explicit no-op skip, sequencing $+\!\!+$ and a typed PUSH; the typing rule for ADD stipulates that the stack layout is correctly set-up for addition, while that for IF expects a Boolean on top of the stack (whose tail has type S), and then executes one of two arbitrary code sequences c_1, c_2 which operate on stacks of type S:

$$\underline{\text{data}} \quad \frac{S, S' \;:\; \mathbf{StackType}}{\mathsf{Code}\, S\, S' \;:\; \star} \quad \underline{\text{where}}$$

$$\frac{}{\mathsf{skip} \;:\; \mathsf{Code}\, S\, S} \qquad \frac{c_1 \;:\; \mathsf{Code}\, S_0\, S_1 \;;\; c_2 \;:\; \mathsf{Code}\, S_1\, S_2}{c_1 +\!\!+ c_2 \;:\; \mathsf{Code}\, S_0\, S_2}$$

$$\frac{v \;:\; \mathbf{Val}\, T}{\mathsf{PUSH}\, v \;:\; \mathsf{Code}\, S\, (T :: S)} \qquad \frac{}{\mathsf{ADD} \;:\; \mathsf{Code}\, (\mathsf{nat} :: \mathsf{nat} :: S)\, (\mathsf{nat} :: S)}$$

$$\frac{c_1, c_2 \;:\; \mathsf{Code}\, S\, S'}{\mathsf{IF}\, c_1, c_2 \;:\; \mathsf{Code}\, (\mathsf{bool} :: S)\, S'}$$

4.4 Implementing an Interpreter for Intermediate Code

Before examining the details in EPIGRAM, we consider the construction of the interpreter **exec** informally. Guided by the above typing rules for intermediate code, it proceeds by case analysis on the code constructor:

case: skip for any stack type S and stack s of that type, return s;

case: $+\!\!+$ this is just iterated composition as usual;

case: PUSH push the corresponding value on the stack at hand;

case: ADD now we can exploit stack typing in earnest: since the input stack s is of type nat :: nat :: S, we know by case analysis on s that it *must* be of the form $n \triangleright m \triangleright s'$ for natural numbers n, m and stack s' (necessarily of type S); this is because the indices occurring in the constructors of the Stack family are all in constructor form, and thus any other stack configuration would be ill-typed (and give rise to a unification failure during type-checking). Thus there is *no* need to explicitly consider ill-typed stacks, *nor* underflow; execution is *guaranteed* to make progress in this case, writing back the natural number $n + m$ on top of s';

case: IF similarly: since the input stack s is of type bool :: S, we know, by case analysis on s that it *must* be of the form $b \triangleright s'$ for some Boolean b and stack s' (necessarily of type S); ditto, by case analysis on b itself, which corresponds to examining the top stack entry, we then jump to the execution of the appropriate branch c_i on stack s', whose type again guarantees, inductively, that execution does not get stuck at this point.

In fact, such an informal analysis usually justifies the stack-safety property, but here provides commentary on the following well-typed piece of EPIGRAM code defining **exec**:

$$\underline{\text{let}} \quad \frac{c \;:\; \mathsf{Code}\, S\, S' \;;\; s \;:\; \mathsf{Stack}\, S}{\mathbf{exec}\; c\; s \;:\; \mathsf{Stack}\, S'}$$

$$\begin{array}{lcc}
\mathbf{exec}\; c\; s \Leftarrow \underline{\text{rec}}\; c & & \\
\quad \mathbf{exec}\, \mathsf{skip}\; s & \Rightarrow & s \\
\quad \mathbf{exec}\, (c_1 +\!\!+\, c_2)\; s & \Rightarrow & \mathbf{exec}\; c_2\, (\mathbf{exec}\; c_1\; s) \\
\quad \mathbf{exec}\, (\mathsf{PUSH}\; v)\; s & \Rightarrow & v \rhd s \\
\quad \mathbf{exec}\; \mathsf{ADD}\; (n \rhd m \rhd s) & \Rightarrow & (n+m) \rhd s \\
\quad \mathbf{exec}\; (\mathsf{IF}\; c_1\, c_2)\; (\mathsf{true} \rhd s) & \Rightarrow & \mathbf{exec}\; c_1\; s \\
\quad \mathbf{exec}\; (\mathsf{IF}\; c_1\, c_2)\; (\mathsf{false} \rhd s) & \Rightarrow & \mathbf{exec}\; c_2\; s
\end{array}$$

4.5 Implementing the Compiler to Intermediate Code

The last piece in the jigsaw is the compiler, **compile**, which we implement via structural recursion, without any further discussion of its type-correctness. Note that we do not need to supply the trailing stack-type S explicitly:

$$\underline{\text{let}} \quad \frac{e \;:\; \mathsf{Exp}\, T}{\mathbf{compile}\, e \;:\; \mathsf{Code}\, S\, (T :: S)}$$

$$\begin{array}{lcl}
\mathbf{compile} \Leftarrow \underline{\text{rec}}\; e & & \\
\quad \mathbf{compile}\; (\mathsf{val}\, v) & \Rightarrow & \mathsf{PUSH}\, v \\
\quad \mathbf{compile}\; (\mathsf{plus}\, e_1\, e_2) & \Rightarrow & (\mathbf{compile}\, e_2) +\!\!+ (\mathbf{compile}\, e_1) +\!\!+ \mathsf{ADD} \\
\quad \mathbf{compile}\; (\mathsf{if}\, b\, e_1\, e_2) & \Rightarrow & (\mathbf{compile}\, b) +\!\!+ \mathsf{IF}\, (\mathbf{compile}\, e_1)\, (\mathbf{compile}\, e_2)
\end{array}$$

5 Compiler Correctness

Equality is a distinguished family in EPIGRAM's type system with one constructor refl. So we may state the correctness property of compilation in its customary equational form, and its proof is simply another dependently-typed functional program, **correct**:

$$\underline{\text{let}} \quad \frac{e \;:\; \mathsf{Exp}\, T \;;\; s \;:\; \mathsf{Stack}\, S}{\mathbf{correct}\, e\, s \;:\; (\mathbf{eval}\, e) \triangleright s \;=\; \mathbf{exec}\, (\mathbf{compile}\, e)\, s}$$

The proof proceeds by induction on the expression, e, and so the implementation of the function proceeds by primitive recursion on e.

case: $\mathsf{val}\, v$ This case is trivial as evaluating the functions on the left and right hand side of the equation both result in pushing the value v onto s. The host-language type of **correct** in this case is computationally equal to $v \triangleright s \;=\; v \triangleright s$, and thus is simply proved by reflexivity, refl.

case: $\mathsf{plus}\, e_1\, e_2$ By induction hypothesis (recursive call on e_1, e_2 respectively), we know that

$$(\mathbf{eval}\, e_1) \triangleright s = \mathbf{exec}\,(\mathbf{compile}\, e_1)\, s;$$
$$(\mathbf{eval}\, e_1) \triangleright s = \mathbf{exec}\,(\mathbf{compile}\, e_1)\, s.$$

Now, the LHS is computationally equivalent to $((\mathbf{eval}\, e_1) + (\mathbf{eval}\, e_2)) \triangleright s$, while the right-hand side becomes $\mathbf{exec}\, \mathsf{ADD}\, ((\mathbf{eval}\, e_2) \triangleright (\mathbf{eval}\, e_1) \triangleright s)$. Finally, the computational rule for $\mathbf{exec}\, \mathsf{ADD}$ finishes the proof, again by reflexivity.

case: $\mathsf{if}\, b\, e_1\, e_2$ As above we have the following induction hypotheses.

$$(\mathbf{eval}\, e_1) \triangleright s = \mathbf{exec}\,(\mathbf{compile}\, e_1)\, s;$$
$$(\mathbf{eval}\, e_1) \triangleright s = \mathbf{exec}\,(\mathbf{compile}\, e_1)\, s;$$

and

$$(\mathbf{eval}\, b) \rhd s = \mathbf{exec}\, (\mathbf{compile}\, b)\, s.$$

Now, by rewriting with this last equation, we reduce the right-hand side to $\mathbf{exec}\, (\mathsf{IF}\, (\mathbf{compile}\, e_1)\, (\mathbf{compile}\, e_2))\, (\mathbf{eval}\, b) \rhd s$, while the left-hand side is just $(\mathbf{eval}\, (\mathsf{if}\ b\ e_1\ e_2)) \rhd s$. We do case analysis on $\mathbf{eval}\, b$ following that of the definition of **eval** — in the true case the problem is solved by the first induction hypothesis, in the false case the problem is solved by the second induction hypothesis (the typing rule for the 'with' program notation in EPIGRAM is precisely designed for this situation where there is a sub-computation, in this case $\mathbf{eval}\, b$ in the definitions of **eval** and **exec**, whose behaviour must be abstracted from its occurrence in a type, namely that of **correct**. The details of this idea are in 'The view from the left' (McBride & McKinna, 2004) section 5).

Note that what we have achieved is a type-correct stratification (at the object-level) of the old compiler correctness diagram. Moreover, the host-level type checking has ensured that the essense of the informal proof (equational reasoning plus appeal to induction hypotheses) is retained in the EPIGRAM implementation of **correct**.

$$\mathsf{Exp}\ T \xrightarrow{\quad\quad\mathbf{eval}\quad\quad} \mathbf{Val}\ T$$

$$\begin{tikzcd} {} \arrow[d, "\mathbf{compile}"'] \arrow[r, phantom, "\mathbf{correct}"] & {} \arrow[d, "\rhd"] \\ \mathsf{Code}\,S\,(T :: S) \arrow[r, "\mathbf{exec}"'] & \mathsf{Stack}\,S \to \mathsf{Stack}\,(T :: S) \end{tikzcd}$$

The code implementing **correct**, the rest of the programs in this paper and EPIGRAM binaries which can be used to execute them, can be found at the above mentioned URL.

6 Conclusion

This paper demonstrates that given a suitably rich host-language type system, exemplified here by EPIGRAM's support for inductive families, important safety properties may be captured entirely by a typed representation of the object-language. Here we have shown two examples of this, namely type-preservation in **eval**, obtained 'for free', while in **exec** we have both stack-type preservation and no stack-underflow. The only correctness property for the compiler which requires separate proof is nevertheless also representable as a host-language program.

It is important to note that the implementations of **eval**, **exec** and **compile** require no annotation to support this correctness proof. What type annotations they may carry are entirely, and largely silently, managed by EPIGRAM's type checker.

Indeed, it is only because type inference is too weak to recognise $n : \mathsf{Nat}$ as an element of $\mathbf{Val}\ T$ for $T = \mathsf{nat}$ which mean we require type tags T at all. It remains an interesting piece of further work to eliminate these tags altogether.

While others have demonstrated the conceptual, theoretical, methodological and practical advantages of maintaining type information throughout compilation from high-level source to assembly language (Morrisett *et al.*, 1999) we hope this paper contributes to the mechanisation of such an approach within an environment such as EPIGRAM.

We gratefully acknowledge our colleagues in the EPIGRAM team, with special thanks to Conor McBride and Peter Morris. Many thanks also go to Graham Hutton, and to the Scottish Programming Languages Seminar. EPIGRAM and this work is generously supported by grants from the EPSRC (grant references GR/N24988, GR/R72259 and EP/C512022).

References

Augustsson, Lennart, & Carlsson, Magnus. (1999). *An exercise in dependent types: A well-typed interpreter*. `http://www.cs.chalmers.se/~augustss/cayenne/`.

Aydemir, Brian E., Bohannon, Aaron, Fairbairn, Matthew, Foster, J. Nathan, Pierce, Benjamin C., Sewell, Peter, Vytiniotis, Dimitrios, Washburn, Geoffrey, Weirich, Stephanie, & Zdancewic, Steve. (2005). Mechanized metatheory for the masses: The PoplMark challenge. *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Lecture Notes in Computer Science. Springer Verlag.

Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**, 440–465.

Gibbons, Jeremy, & de Moor, Oege (eds). (2003). *the fun of programming*. Palgrave. Chap. 12.

Goguen, Healfdene, McBride, Conor, & McKinna, James. (2006). Eliminating dependent pattern matching. Futatsugi, Koichi, Jouannaud, Jean-Pierre, & Meseguer, José (eds), *Algebra, Meaning and Computation: a 65th birthday volume for Joseph Goguen*. Lecture Notes in Computer Science, vol. 4060. Springer-Verlag.

McBride, Conor. (2002). Faking It (Simulating Dependent Types in Haskell). *Journal of functional programming*, **12**(4& 5), 375–392. Special Issue on Haskell.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of Functional Programming*, **14**(1), 69–111.

McCarthy, John, & Painter, James. (1967). Correctness of a compiler for arithmetic expressions. *Pages 33–41 of: Proceedings of the XIXth AMS Symposium on Applied Mathematics, Mathematical Aspects of Computer Science.*

Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, **21**(3), 527–568.

Pollack, Randy. (1990). *Implicit syntax.* In: Preliminary Proceedings of the 1st Workshop on Logical Frameworks, http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc90.ps.gz, pages 421–433.