Effect Inference for Deterministic Parallelism

Karl-Filip Faxén
Swedish Institute of Computer Science
kff@sics.se

April 10, 2008

SICS Technical Report T2008:08, ISSN 1100-3154

Abstract

In this report we sketch a polymorphic type and effect inference system for ensuring deterministic execution of parallel programs containing shared mutable state. It differs from that of Gifford and Lucassen in being based on Hindley Milner polymorphism and in formalizing the operational semantics of parallel and sequential computation.

Keywords: Effect inference, type inference, parallel execution, operational semantics, side effects, polymorphism

1 Introduction

With the advent of multicore processors, all programming is parallel programming. The standard way to meet this requirement is to use a conventional imperative sequential language extended with some form of support for parallelism, ranging from a pure library solution like `pthreads` over annotations like OpenMP to language extensions such as those in Cilk [2]. In each case, the parallel activities share state and accesses to that state need to be explicitly synchronized to avoid race conditions.

In general, this will lead to a semantics that is not confluent, that is, different evaluation orders can give different results. This parallels the situation when writing explicitly parallel code in real languages; programs are nondeterministic in general, and unless great care is taken some of the possible behaviors are undesirable, crashing or deadlocking the program or, even worse, making it behave subtly different from its specification.

Hence confluence, or determinacy, is a desirable property. One alternative is to use languages where programs are confluent by construction, for instance functional languages. There are however limitations to the applicability of this approach; converting legacy code to a functional language is often prohibitively expensive, and they also have performance and resource control problems.

This paper follows the approach pioneered by Lucassen and Gifford [6] by presenting a different approach, staying within the imperative world of programming with side effects, but using a type system for taming these and ensure confluence. In this paper we present a type and effect inference system for a simple lambda calculus with explicit parallelism and side effecting operations. We believe that the ideas carry over to more conventional languages; in recent years innovations in type systems have been added to conventional base languages, for instance in Cyclone [4] and Pizza [8]. Indeed, templates in C++ and generics in Java are also examples.

2 The Language

We use a call-by-value lambda calculus extended with updatable references. The syntax is given in Figure 1.

$e \in \text{Expr}$	$\rightarrow$	$x \mid \lambda x.e \mid e_1\ e_2$
	$\mid$	$c \mid e_1 \mid e_2 \mid \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2$
$v \in \text{Value}$	$\rightarrow$	$\lambda x.e \mid a \mid c$
$E \in \text{EvalCtx}$	$\rightarrow$	$\Box\ e \mid v\ \Box \mid \texttt{let}\ x{=}\Box\ \texttt{in}\ e$
	$\mid$	$\Box \mid e \mid e \mid \Box$
$c \in \text{Const}$	$\rightarrow$	$\texttt{new} \mid \texttt{get} \mid \texttt{set} \mid \texttt{rec} \mid () \mid \texttt{1} \mid \ldots$

Figure 1: The language

The operator `new` allocates a new reference cell and initializes it to the value of its argument while `get` and `set` denote dereference and update of reference cells, respectively. The syntax also contains parallel compositions of the form $e_1|e_2$ where the expressions are evaluated in parallel for their side effects. Let expressions provide a means of sequencing; we will use $e_1;e_2$ as a shorthand for $\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2$ where x is not free in e_2.

The semantics, given in figure 2, is a small step operational semantics defined using evaluation contexts. It consists of rules for proving that a configuration H, e, where H is a *heap* mapping addresses to values, rewrites in one step to H', e'. Expressions are extended with addresses, ranged over by a, which must be bound by the heap. The [join] rule provides synchronization at the completion of evaluation of a parallel composition while the fact that the branches can not be reduced until the whole expression is to be reduced provides synchronization of the start. Thus the parallelism in the language follows the fork/join model.

An evaluation context E is an "expression with a hole"; it contains exactly one occurrence of the symbol $\Box$, and for any expression e, $E[e]$ is E with $\Box$ replaced by e. The hole in an evaluation context marks the immediate subexpression within which the next reduction step should be taken. For example, when reducing an application, the function part is reduced first as indicated by the evaluation context $\Box\ e$. When the function is a value, the argument is reduced ($v\ \Box$). In contrast, either branch in a parallel composition can be reduced (replacing $e|\Box$ with $v|\Box$ would force sequential left to right evaluation).

$$H,\ (\lambda x.e)\ v \longrightarrow H,\ [x \leftarrow v]e \qquad \text{app}$$

$$H,\ ()|() \longrightarrow H,\ () \qquad \text{join}$$

$$H,\ \texttt{let}\ x{=}v\ \texttt{in}\ e \longrightarrow H,\ [x \leftarrow v]e \qquad \text{let}$$

$$H,\ \texttt{new}\ v \longrightarrow H[a \mapsto v],\ a \qquad \text{new}$$

$$H,\ \texttt{get}\ a \longrightarrow H,\ H(a) \qquad \texttt{get}$$

$$H,\ \texttt{set}\ a\ v \longrightarrow H[a \mapsto v] \qquad \texttt{set}$$

$$H,\ \texttt{rec}\ \lambda x.e \longrightarrow [x \leftarrow \texttt{rec}\ \lambda x.e]e \qquad \text{rec}$$

$$\frac{H,\ e \longrightarrow H',\ e'}{H,\ E[e] \longrightarrow H', E[e']} \qquad \text{ctx}$$

Figure 2: Operational semantics

2.1 Confluence

The combination of parallelism and side effects make the semantics non confluent, as demonstrated by the example in figure 3 where the normal form of an expression depends on which of the branches is evaluated first. After the first rewrite steps, either the left branch is reduced first (middle section in the figure), yielding 2 as normal form, or the right branch can be reduced first (last part) resulting in 1.

The lack of confluence of the evaluation relation is easy to fix by choosing the above mentioned sequential semantics for parallel composition. This yields a semantics where every reducible term has a single redex. We formalize this as a sequential evaluation relation $\longrightarrow_S$.

Definition 1 (Sequential evaluation) *We define the* sequential evaluation *relation* $\longrightarrow_S$ *using the derivation rules for the evaluation relation* $\longrightarrow$ *in figure 2 with the difference that the alternative* $e|\Box$ *in*

$[]$, `let c =` `new 0 in` (`set c 1`|`set c 2`); `get c`
$\longrightarrow [a_0 \mapsto 0]$, `let c =`$a_0$ `in` (`set c 1`|`set c 2`); `get c`
$\longrightarrow [a_0 \mapsto 0]$, (`set` a_0 `1`|`set` a_0 `2`); `get` a_0
$\longrightarrow [a_0 \mapsto 1]$, (()|`set` a_0 `2`); `get` a_0
$\longrightarrow [a_0 \mapsto 2]$, (()|()); `get` a_0
$\longrightarrow [a_0 \mapsto 2]$, (()); `get` a_0
$\longrightarrow [a_0 \mapsto 2]$, `get` a_0
$\longrightarrow [a_0 \mapsto 2]$, `2`
$\longrightarrow [a_0 \mapsto 2]$, (`set` a_0 `1`|()); `get` a_0
$\longrightarrow [a_0 \mapsto 1]$, (()|()); `get` a_0
$\longrightarrow [a_0 \mapsto 1]$, (()); `get` a_0
$\longrightarrow [a_0 \mapsto 1]$, `get` a_0
$\longrightarrow [a_0 \mapsto 1]$, `1`

Figure 3: An example illustrating lack of confluence

the definition of evaluation context is replaced by $v|\Box$ *so that the left branch is always evaluated first.*

The point of the sequential evaluation relation is that it is confluent, even deterministic. To state the lemma we need to define configurations that are essentially the same.

Definition 2 *Two configurations are* α*-equivalent, written* $H_1, e_1 \approx H_2, e_2$*, if they are equivalent up to renaming of addresses.*

Note that $\approx$ is an equivalence relation. In particular, any configuration is α-equivalent to itself. Our confluence lemma states that sequential evaluation preserves α-equivalence.

Lemma 1 *If* $H_1, e_1 \longrightarrow_S H_1', e_1'$ *and* $H_2, e_2 \longrightarrow_S H_2', e_2'$ *and* $H_1, e_1 \approx H_2, e_2$*, then* $H_1', e_1' \approx H_2', e_2'$*.*

Proof: By induction on e_1. Omitted. □

As a simple corollary, the lemma holds for arbitrary finite sequences of reduction steps.

3 Type system

The system presented here is essentially that of ML [7]; we have a call-by-value lambda calculus with updatable references and let-polymorphism. Side effects are captured in the type system using *effects*, which keep track of the fact that evaluation of an expression might read or write values to the heap. The typing rule for parallel composition checks that the effects of the parallel branches are compatible: If one of them writes a heap location, the other does not access it. To distinguish different heap locations we use *regions* which represent sets of heap locations and we give rules for proving that two regions are disjoint.

We present the syntax of types in our system in Figure 4. A (monomorphic) type is either a type variable α, a data type T instantiated with types,

$\tau \in$ Type	$\rightarrow$	$\alpha \mid T\ \bar{\tau}\ \bar{\rho}\ \bar{\kappa} \mid \tau_1 \xrightarrow{\kappa} \tau_2 \mid \mathtt{ref}\,\rho\,\tau$
$\kappa \in$ Effect	$\rightarrow$	$\eta \mid \mathtt{R}\,\rho \mid \mathtt{W}\,\rho \mid \kappa_1 \cup \kappa_2 \mid \epsilon$
$\rho \in$ Region	$\rightarrow$	γ
$\sigma \in$ Scheme	$\rightarrow$	$\forall\bar{\alpha}\bar{\gamma}\bar{\eta}.\Phi \Rightarrow \tau$
$\Phi \in$ Constraint	$\rightarrow$	$\kappa_1 \mathrel{\mathtt{<}} \kappa_2 \mid \rho_1 \mathrel{\mathtt{<}} \rho_2 \mid \Phi_1 \wedge \Phi_2 \mid \mathbf{tt}$

$\alpha \in$ TyVar, $\eta \in$ EffVar, $\gamma \in$ RgnVar, $T \in$ TypeName

Figure 4: Types in the system

$$\Phi_1 \wedge \Phi_2 \vdash \Phi_1 \qquad \text{PROJ}$$

$$\frac{\Phi \vdash \rho_1|\rho_2}{\Phi \vdash \mathtt{W}\,\rho_1|\mathtt{W}\,\rho_2} \qquad \text{WW}$$

$$\frac{\Phi \vdash \rho_1|\rho_2}{\Phi \vdash \mathtt{W}\,\rho_1|\mathtt{R}\,\rho_2} \qquad \text{RW}$$

$$\Phi \vdash \mathtt{R}\,\rho_1|\mathtt{R}\,\rho_2 \qquad \text{RR}$$

$$\frac{\Phi \vdash \kappa_1|\kappa_3 \quad \Phi \vdash \kappa_2|\kappa_3}{\Phi \vdash (\kappa_1 \cup \kappa_2)|\kappa_3} \qquad \text{DIST}$$

Figure 5: Constraint entailment

regions and effects or a function type $\tau_1 \xrightarrow{\kappa} \tau_2$ from types τ_1 to τ_2 with latent effect κ. The latent effect of a function type is the effect of calling the function, that is, the effect of the function body. An effect is either an effect variable η, a read effect $\mathtt{R}\,\rho$, a write effect $\mathtt{W}\,\rho$ or a combined effect $\kappa_1 \cup \kappa_2$. In this system, regions are only region variables γ.

Effects and regions are used to build constraints which give the requirements for a parallel composition to be safe. The compatibility constraint $\kappa_1|\kappa_2$ is satisfied if no region that occurs in a write effect in one of the κ_i occurs in any effect in the other. This can be reduced to pairwise disjointness constraints $\rho_1|\rho_2$ on the regions involved. These are satisfied if the regions are distinct region variables. We also have conjunction of constraints; $\Phi_1 \wedge \Phi_2$ is satisfied if Φ_1 and Φ_2 are satisfied. We treat $\wedge$ as associative and commutative, that is $(\Phi_1 \wedge \Phi_2) \wedge \Phi_3 = \Phi_1 \wedge (\Phi_2 \wedge \Phi_3)$ and $\Phi_1 \wedge \Phi_2 = \Phi_2 \wedge \Phi_1$.

Figure 5 formalizes this intuition by defining an *entailment* relation between constraints. Some constraints are always satisfied; for instance two reads never conflict, which is the motivation for the rule [RR] which allows to prove for instance $\mathbf{tt} \vdash \mathtt{R}\,\rho_1|\mathtt{R}\,\rho_2$ for any regions ρ_1 and ρ_2($\mathbf{tt}$ is the always satisfied constraint which can be seen as an empty conjunction). For other constraints, satisfaction depends

$$\begin{aligned}
&\vdash \mathtt{new} : \tau \xrightarrow{\epsilon} \mathtt{ref}\,\rho\,\tau \\
&\vdash \mathtt{get} : \mathtt{ref}\,\rho\,\tau \xrightarrow{\mathtt{R}\,\rho} \tau \\
&\vdash \mathtt{set} : \mathtt{ref}\,\rho\,\tau \xrightarrow{\epsilon} \tau \xrightarrow{\mathtt{W}\,\rho} () \\
&\vdash \mathtt{rec} : (\tau \xrightarrow{\kappa} \tau) \xrightarrow{\kappa} \tau \\
&\vdash () : () \\
&\vdash \mathtt{1} : \mathtt{Int} \\
&\ldots
\end{aligned}$$

Figure 7: Types of the builtin operations

on the parts. For instance, the conflict constraint $\mathtt{W}\,\rho_1|\mathtt{R}\,\rho_2$ is satisfied if the disjointness constraint $\rho_1|\rho_2$ is satisfied (rule [RW]). A constraint also entails its parts (the [PROJ] rule, where the simple formulation relies on the associativity and commutativity of $\wedge$). Note that there is no rule for entailing disjointness constraints except by projection. The idea is that $\rho_1|\rho_2$ is satisfied if ρ_1 and ρ_2 are distinct region variables, but if a rule to that effect is included, we would lose the property that entailment is closed under substitution. For example, we would have $\mathbf{tt} \vdash u|v$ but substituting u for v would yield $u|u$ which is not entailed by $\mathbf{tt}$.

A type scheme σ of the form $\forall\bar{\alpha}\bar{\gamma}\bar{\eta}.\Phi \Rightarrow \tau$ represents all substitution instances $\theta\tau$ such that $\theta\Phi$ is satisfied and θ is of the form $[\bar{\tau}/\bar{\alpha}, \bar{\rho}/\bar{\gamma}, \bar{\kappa}/\bar{\eta}]$.

The inference rules presented in figure 6 allows to prove typing judgments of the form $A, \Phi \vdash e : \sigma\&\kappa$ where A gives (polymorphic) types to the free variables of e, σ is its (polymorphic) type and κ is the effect of evaluating e. Figure 7 gives the typing rules for constants. Unsurprisingly, these do not depend on the typing assumptions for the variables, and they are also independent of the constraints. Since constants are not evaluated, they have no side effects, but `get` and `set` have latent effects (in fact, all effects in the system come ultimately from these).

Figure 8 gives an example of how the type system captures aliasing constraints in the types. Here is a function taking two reference cells as arguments and writing integers into them in parallel. This is only confluent if the cells are different, which is captured

$$\frac{A(x) = \sigma}{A, \Phi \vdash x : \sigma \& \epsilon} \quad \text{VAR}$$

$$\frac{A, \Phi \vdash e_1 : \tau_1 \xrightarrow{\kappa} \tau_2 \& \kappa_1 \quad A, \Phi \vdash e_2 : \tau_1 \& \kappa_2}{A, \Phi \vdash e_1\ e_2 : \tau_2 \& \kappa \cup \kappa_1 \cup \kappa_2} \quad \text{APP}$$

$$\frac{A[x \mapsto \tau_1], \Phi \vdash e : \tau_2 \& \kappa}{A, \Phi \vdash \lambda x.e : \tau_1 \xrightarrow{\kappa} \tau_2 \& \epsilon} \quad \text{ABS}$$

$$\frac{\vdash c : \tau}{A, \Phi \vdash c : \tau \& \epsilon} \quad \text{CON}$$

$$\frac{A, \Phi \vdash e_1 : \sigma \& \kappa_1 \quad A[x \mapsto \sigma], \Phi \vdash e_2 : \tau \& \kappa_2}{A, \Phi \vdash \texttt{let}\ x{=}e_1\ \texttt{in}\ e_2\ : \tau \& \kappa_1 \cup \kappa_2} \quad \text{LET}$$

$$\frac{A, \Phi \vdash e_1 : () \& \kappa_1 \quad A, \Phi \vdash e_2 : () \& \kappa_2 \quad \Phi \vdash \kappa_1 | \kappa_2}{A, \Phi \vdash e_1 | e_2 : () \& \kappa_1 \cup \kappa_2} \quad \text{PAR}$$

$$\frac{A, \Phi' \vdash v : \tau \& \kappa \quad \bar{\alpha}, \bar{\gamma}, \bar{\eta} \cap \mathrm{fv}(A, \kappa) = \emptyset}{A, \Phi \vdash v : (\forall \bar{\alpha} \bar{\gamma} \bar{\eta}. \Phi' \Rightarrow \tau) \& \kappa} \quad \text{GEN}$$

$$\frac{A, \Phi \vdash e : (\forall \bar{\alpha} \bar{\gamma} \bar{\eta}. \Phi' \Rightarrow \tau) \& \kappa \quad \theta = [\bar{\tau}/\bar{\alpha}, \bar{\rho}/\bar{\gamma}, \bar{\kappa}/\bar{\eta}]}{A, \Phi \wedge \theta\Phi' \vdash e : \theta\tau \& \kappa} \quad \text{INST}$$

Figure 6: The dependence type system

in the constraint $u|v$ in the type scheme. This easy way of summarizing information about procedures is one of the major appeals of using type inference as a framework for what is traditionally done with more ad-hoc techniques.

3.1 Soundness

In this section we deal with the correctness of the type system. Does it achieve what we advertise? Are well typed programs guaranteed to be confluent? We will approach this question in several steps. First, we will prove that evaluation preserves typing; this is known as a *subject reduction lemma*. To be able to do this, we extend the type system to type the configurations that occur in the operational semantics. These differ from expressions in that they extend the expression syntax with addresses (ranged over by a) and heaps that bind the addresses. We do this in a way that makes the expression typing agree with the configuration typing when applied to a configuration with an empty heap an an expression not containing addresses.

Next, we prove that if a configuration is typable then it is either a value or it can be rewritten using some rule in the operational semantics; a *progress lemma*. This is the classical soundness result for the kind of small step rewrite semantics we use.

Then we come to the result that is at the heart of our system. Recall that the language we have defined in section 2 is not confluent in general. That is, depending on the interleaving of the reduction steps in the two parts of a parallel composition, an expression can reduce to different values. We now claim that if a parallel composition is well-typed, the interleaving of evaluation steps can be changed without affecting the result. This means that all interleavings give the same result, establishing confluence since the rest of the semantics is confluent.

The following definition extends type inference from expressions to stores (heaps).

Definition 3 (Store typing) *We extend typing assumptions A to map addresses to types of the form* $\texttt{ref}\,\gamma\,\tau$ *and we write $\Phi \vdash H : A$ if for every address a in the domain of A, $A(a) = \texttt{ref}\,\gamma\,\tau$ for some γ and τ such that $A, \Phi \vdash H(a) : \tau\&\epsilon$.*

Now that we can do type inference for heaps and expressions, we can infer types for configurations.

Definition 4 (Configuration typing) *If $A, \Phi \vdash e : \tau\&\kappa$ and $\Phi \vdash H : A$ we say that the configuration H, e has type τ and effect κ under the assumptions A, Φ, written $A, \Phi \vdash H, e : \tau\&\kappa$.*

Most polymorphic type systems have some form of substitution lemma since it underlies the proof of soundness for generalization/instantiation. The lemma states that if a particular derivation can be made, then any substitution instance of the derivation is also legal.

Lemma 2 (Substitution) *If $A, \Phi \vdash e : \tau\&\kappa$, then for any substitution θ, $\theta A, \theta\Phi \vdash e : \theta\tau\&\theta\kappa$.*

Finally we arrive at the subject reduction lemma.

Lemma 3 (Subject reduction) *If $A, \Phi \vdash H, e : \tau\&\kappa$ and $H, e \longrightarrow H', e'$ then there is an $A' \supseteq A$ (possibly extending A with a new address) such that $A', \Phi \vdash H', e' : \tau\&\kappa$.*

Proof: Omitted. □

The next step in our correctness arguments is the progress lemma, stating that in a well typed configuration, the expression is either a value (in which case evaluation has terminated) or another evaluation step can be taken.

Lemma 4 (Progress) *If $.A, \Phi \vdash H, e : \tau\&\kappa$ then either e is a value or there is a configuration H', e' such that $H, e \longrightarrow H', e'$.*

Proof: Omitted. □

We finally arrive at the main semantic equivalence theorem, which states that if a well typed configuration is rewritten in a finite number of steps to a value, then that configuration may be rewritten to an equivalent value using the sequential evaluation relation.

$$\lambda \mathtt{x}.\lambda \mathtt{y}.(\mathtt{set\ x\ 1}|\mathtt{set\ y\ 2}) : \forall u\ v.(u|v) \Rightarrow \mathtt{ref}\ u\ \mathtt{Int} \xrightarrow{\epsilon} \mathtt{ref}\ v\ \mathtt{Int}\ \overset{\mathtt{W}\,u \cup \mathtt{W}\,v}{\rightarrow}\ ()$$

Figure 8: Aliasing constraints in types

Theorem 1 (Confluence) *If* $A, \Phi \vdash H, e : \tau \& \kappa$, Φ *is satisfiable and* $H, e \longrightarrow^* H_1, v_1$ *then there are* H_2, v_2 *such that* $H, e \longrightarrow_S^* H_2, v_2$ *and* $H_1, v_1 \approx H_2, v_2$.

Proof: Omitted. □

4 Related work

Gifford and Lucassen [6] pioneered the use of effect inference for ensuring confluence of parallel programs containing shared mutable state. Subsequently, Jouvelot and Gifford gave a type inference algorithm, but only for an effect system without regions [5].

Our approach is somewhat similar in spirit to automatic parallelization since the confluent semantics of well typed programs in fact coincides with a purely sequential semantics [3, 1]. The difference is that parallelism is explicit in our model, and we think that this is essential in the long run. Clearly, parallel programs do not just happen. On the contrary, parallelism must be designed into the program just like type correctness is designed into a program in a strongly typed language. We regard an unsafe explicit parallel construct as an error in the program whereas in automatic parallelization the corresponding situation, a construct meant to be parallelizable but which is not, silently yields degraded performance.

5 Future work

This paper is based on a very simple type system. To achieve better precision, there are other type systems that could be used. In particular more powerful forms of polymorphism allows the typing of more programs, as does the use of intersection types.

The present work also considers each heap cell as a unit; the entire cell is read or written. This is too coarse for many array intensive programs; in general accesses to different elements of an array needs to be disambiguated. To formalize this, it seems suitable to use *dependent types*, a discipline where types can be indexed by values. In this case it appears promising to have indexed regions. For such regions, read and write effects on the same regions would not conflict if their indices could be proved to be always different. We believe that such a type system could be checked using techniques very similar to those already used in automatically parallelizing compilers.

References

[1] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, January 2008.

[2] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[3] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[4] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX An-*

nual Technical Conference, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[5] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–310, New York, NY, USA, 1991. ACM.

[6] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.

[7] Robin Milner, Mads Tofte, Robert Harper, and David B. MacQueen. *The Definition of Standard ML, (Revised)*. MIT Press, 1997.

[8] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, New York, NY, USA, 1997. ACM.