Deriving a Strongly Normalizing STG Machine

Dirk Kleeblatt

Technische Universität Berlin
Fakultät IV, Sekr. TEL 12-2
Ernst-Reuter-Platz 7
D-10587 Berlin
klee@cs.tu-berlin.de

Abstract: We present a modified version of the spineless tagless graph machine (or STG machine for short), that can deal with free variables and makes it possible to use compiled code for the normalization of functional expressions. We derive the machine from a high level semantics, thus enabling a simple correctness proof. Our modified STG machine has been successfully implemented in the Ulysses system.

1 Introduction

Traditionally, compiled systems compute so called *weak head normal forms* or WHNF for short, i. e. they perform no computations beneath λ-abstractions or case analyses. For some applications, this is not enough: for example in dependent type checking, *strong* normal forms are required that must not contain any redexes to test for β-convertibility. This requires to deal with free variables since e. g. formal parameters of abstractions occur free in the body of the abstraction, and compiled code usually cannot deal with this free variables. Therefore, dependent type checkers usually include interpreters to reduce type expressions containing arbitrary user defined functions. But this has several disadvantages:

- Interpreted code has reduced performance compared to compiled code.
- When writing a compiler, an additional interpreter is needed just for type checking, and when the language is extended later on, two different parts of code have to be adapted: the compiler as well as the interpreter.
- This gets worse in the presence of even small differences in the semantics of the interpreter and the compiler: computations giving a different result at runtime than during type checking will most probably violate type safety.

We present a strong normalization system for FUN, a simple lazy functional language, that overcomes this disadvantages by using compiled code for normalization and is well suited for dependent type checking. The generated code is efficient enough and can directly be used as the final compiler output if type checking is successful. We start with a big step operational semantics, that can easily be implemented by an interpreter, and transform it in several steps to an abstract machine executing pseudo-assembler code. The present work is a formalization of the implementation of Ulysses, a dependently typed lazy functional language informally introduced in [Kle08], that goes even one step further: instead of pseudo-assembler code, Ulysses generates x86 machine code. Hence it is a convincing example of the applicability of the systems presented here.

The original definition of FUN, its evaluation to WHNF, and the derivation of a compiled system is due to [dlEP03]. We assume some familiarity with this work, since our S4, SSTG1, SSTG2 and ISSTG resemble their S3, STG1, STG2 and ISTG. We extend the weak evaluator with *accumulators* representing irreducible expressions that contain free variables at redex positions, and a read back phase. Accumulators and read back were introduced by [GL02] in a strong normalization system for a strict language.

In the following, we present the language FUN (section 2), give a big step semantics (section 3), derive linearized versions (sections 4 and 5) and show how machine code can be generated (section 6), before giving a comparison with related work (section 7).

2 The Language FUN and its Normal Forms

The syntax of FUN and its normal forms is given in figure 1. It is intended as a intermediate language due to its restrictions. Here and in the following boldface font signifies sequences of objects, i. e. while x is a variable, $\boldsymbol{x}$ is a sequence of variables. We use the sequence operations $\cdot\!:$, $:\!\cdot$ and $+\!\!+$ for prepending and appending a single element and concatenation, respectively. In parameter lists of applications and abstractions we express concatenation by juxtaposition. The empty sequence is denoted by $\Diamond$. We make use of unspecified syntactical categories x of variables, p of pointers and C of constructors.

In the simplest case a FUN expression e is a reference, i. e. either a variable x or an pointer to the heap p. A description of Heaps is given below. Function application is restricted to references in argument position, but more than one may be given as argument. Variables can be bound to so-called λ-forms lf with `let`, where the right hand side might be another expression, a constructor application that has to be saturated, or a lambda abstraction that may abstract over several variables at once. Local bindings are the only places where constructors and abstractions may occur. Definition by cases can be done for arbitrary expressions. A sequence of alternatives $\boldsymbol{alt}$ relates constructors to expressions, binding the constructor arguments to variables (if any).

A normal form v is either an application of an irreducible head h to a sequence of normal forms (that may be empty), a constructor with normal forms in argument position or a λ-abstraction with a normal form in its body. An irreducible head h is either a plain variable or a case discrimination where the scrutinee is not a constructor but an application of another irreducible head to zero or more normal forms.

In normal forms, function and constructor arguments are not restricted to be variables. Furthermore, λ-abstractions and constructors are not restricted to right hand sides of local bindings. Another restriction is added to normal forms that is not applied to FUN expres-

$$
\begin{array}{lcl}
e & ::= & \mathit{ref} \\
  & \mid & e\ \boldsymbol{ref} \\
  & \mid & \texttt{let}\ \boldsymbol{x} = \boldsymbol{lf}\ \texttt{in}\ e \\
  & \mid & \texttt{case}\ e\ \texttt{of}\ \boldsymbol{alt} \\
\\
\mathit{ref} & ::= & x \mid p \\
\\
\mathit{lf} & ::= & e \\
  & \mid & C\ \boldsymbol{x} \\
  & \mid & \lambda\, \boldsymbol{x}\,.\, e \quad (|\boldsymbol{x}| > 0) \\
\\
\mathit{alt} & ::= & C\ \boldsymbol{x}\ \texttt{->}\ e
\end{array}
\qquad
\begin{array}{lcl}
v & ::= & h\,\boldsymbol{v} \\
  & \mid & C\,\boldsymbol{v} \\
  & \mid & \lambda\, x\,.\, v \\
\\
h & ::= & x \\
  & \mid & \texttt{case}\ h\,\boldsymbol{v}\ \texttt{of}\ \boldsymbol{valt} \\
\\
\mathit{valt} & ::= & C\ \boldsymbol{x}\ \texttt{->}\ v
\end{array}
$$

Figure 1: Syntax of FUN and its normal forms

sions. Abstractions may abstract over several variables at once in expressions, but not in the normal forms. While the abstract machines defined in the following chapters can enhance efficiency by considering multiple abstracted variables in one step, such considerations are not important for normal forms. Additionally, we can simplify our presentation a little bit by restricting abstractions in normal forms to single variables.

3 A Big Step Semantics for FUN: S4

We first give a big step operational semantics, called S4 to express its relationship to S3 from [dlEP03]. Since we want to employ lazy evaluation, we have to implement *sharing* to avoid duplicated evaluations of subterms. Thus, we need a *heap*, i. e. a mapping form pointers to heap values hv, according to the grammar in figure 2.

We allocate each expression that is bound by `let` on the heap, hence the first kind of heap values are λ-forms. When an unevaluated expression allocated on the heap (a so-called *thunk*) is evaluated, it is overwritten by an indirection $\looparrowright q$ to its WHNF at address q.

In addition, we need a representation of free variables and other subexpression that are irreducible because they contain free variables at redex positions. Free variables are inserted to evaluate expressions under λ-abstractions and case discriminations. When such a free variable is used in function position or as a scrutinee of a case discrimination, it collects the context of this occurrence for later analysis. Thus, these special kind of heap value is called *accumulator*, denoted by the meta variable k .

The structure of accumulators is also given in figure 2. An accumulator k is either a free variable, or a suspended application of another accumulator to a single argument pointer, or a suspended case discrimination with another accumulator as scrutinee. To maintain sharing, we only reference embedded accumulators by pointers. We ensure the invariant

$$
\begin{array}{lcl}
hv & ::= & lf \\
 & \mid & \looparrowright p \\
 & \mid & k
\end{array}
\qquad\qquad
\begin{array}{lcl}
k & ::= & \langle x \rangle \\
 & \mid & \langle p\, q \rangle \\
 & \mid & \langle \texttt{case}\ p\ \texttt{of}\ \boldsymbol{alt} \rangle
\end{array}
$$

Figure 2: Heap Values and Accumulators for S4

$$
Norm\ \frac{\Gamma : e \downarrow \Delta : p \qquad \Delta : p \uparrow \Theta : v}{\Gamma : e \updownarrow \Theta : v}
$$

Figure 3: Normalization by Weak Evaluation and Read Back

that the heap values referenced in function and scrutinee position are accumulators in all inference rules of our semantics. To make the difference between accumulators and usual expressions visible, we enclose accumulators in angle brackets.

We use the following notations for expressing heap allocations: $\Gamma[p \mapsto hv]$ means that the heap value hv is bound to address p in Γ, so no new allocation takes place here. $\Gamma \cup [p \mapsto hv]$ means that a new binding is added to Γ, thus p is not bound to any heap value in Γ itself, but is bound to hv in the resulting new heap. Here, a new allocation is performed. Additionally, we use a special binding $\Gamma \Mapsto hv$ for heap values that need no updates after normalization to WHNF.

3.1 Normalizing FUN expressions

The normalization of FUN expressions is done with two interacting systems. An evaluator reduces an expression to WHNF, using a modified version of the STG machine, originally defined in [PJ92]. The resulting WHNF is *read back*, normalizing all remaining redexes.

Thus, the overall system is defined using three relations. *Weak evaluation* is defined by the four place relation $\cdot : \cdot \downarrow \cdot : \cdot$, where $\Gamma : e \downarrow \Delta : p$ means that expression e evaluates under heap Γ to the new heap Δ that contains the WHNF of e at address p. *Read back* is defined by the four place relation $\cdot : \cdot \uparrow \cdot : \cdot$, where $\Gamma : p \uparrow \Delta : v$ means that reading back the WHNF located in Γ at address p yields the new heap Δ and the strong normal form v. *Strong normalization* is defined via weak evaluation and read back as the four place relation $\cdot : \cdot \updownarrow \cdot : \cdot$, where $\Gamma : e \updownarrow \Delta : v$ means that the normal form of e under the heap Γ is v, and the normalization yields the new heap Δ.

The normalization relation is inductively defined by the single inference rule given in figure 3 as the composition of weak evaluation and read back, which are defined below.

Weak Evaluation The evaluation relation is defined inductively by the inference rules in figure 4. Rules Lam through $Case_1$ are the same as in the semantics S3 defined by [dlEP03], where a description can be found. We introduced rules $Accu$, App_3 and $Case_2$ to deal with accumulators. They are especially designed for cooperation with read back.

The three new rules define the behavior of accumulators in different contexts. Rule $Accu$ defines an accumulator as a new kind of WHNF, no further evaluation is required.

More interesting is rule App_3. When an expression e in function context is evaluated to an accumulator k, the accumulator grabs the first argument, and the accumulated application is allocated at a fresh address r and applied to the remaining arguments.

The corresponding situation for case discriminations is found in rule $Case_2$. When the scrutinee is evaluated to an accumulator, no branch of the alternatives can be selected, hence the discrimination is suspended in a newly allocated accumulator, saving all branches for further analysis by the read back phase.

Read Back The read back relation is inductively defined by the rules in figure 5. It makes use of a four place helper relation $\cdot : \cdot \Uparrow_{\langle\rangle} \cdot : \cdot$ for read back of accumulators.

Rule $Lam_\Uparrow$ defines read back for λ-abstractions. To evaluate the body under the abstraction, we generate a pseudo-argument: a fresh variable placed as an accumulator onto the heap. The application of the abstraction to this accumulator is normalized to a value v that now contains y as a free variable. To close the resulting expression, we place v below an abstraction over y in the conclusion.

The read back for constructor values is defined by rule $Cons_\Uparrow$. We simply normalize all constructor arguments one at a time. The notation $\bigwedge_{i=1}^{|\boldsymbol{q}|}$ is a shorthand to define the individual premises needed for each constructor argument.

Accumulators are delegated to the helper relation by rule $Accu_\Uparrow$. Free variables need no further normalization, as stated by rule $Var_{\Uparrow_{\langle\rangle}}$.

Suspended applications are read back by reading back the function part and (pointing always to an accumulator, as stated above), and normalizing the argument. Note that all rules for the read back of accumulators result in a normal form with a shape like $h\boldsymbol{v}$, that is in an application of a head h to zero or more normal forms $\boldsymbol{v}$.

Suspended case discriminations are handled by rule $Case_{\Uparrow_{\langle\rangle}}$. The scrutinee is read back to value v. Moreover, for each alternative, a new constructor value $C_i\,\boldsymbol{q}$ is allocated, where the constructor arguments are pointers to fresh free variable accumulators. Then, this constructor expressions are scrutinized, thereby passing the accumulators $\langle\boldsymbol{y}\rangle$ to the corresponding branch of the alternative that now can be evaluated.

4 Linearization: SSTG1

To substitute the interpreted evaluation using S4 with a compiled system, we next define SSTG1, short for strong normalizing STG machine, a small step operational semantics

$$\mathit{Lam}\,\frac{}{\Gamma[p \mapsto \lambda\,\boldsymbol{x}\,.\,e] : p \;\Downarrow\; \Gamma : p}$$

$$\mathit{Cons}\,\frac{}{\Gamma[p \mapsto C\,\boldsymbol{p'}] : p \;\Downarrow\; \Gamma : p}$$

$$\mathit{Var}\,\frac{\Gamma : e \;\Downarrow\; \Delta : q}{\Gamma \cup [p \mapsto e] : p \;\Downarrow\; \Delta \cup [p \mapsto\!\!\!\looparrowright q] : q}$$

$$\mathit{Ind}\,\frac{}{\Gamma[p \mapsto\!\!\!\looparrowright q] : p \;\Downarrow\; \Gamma : q}$$

$$\mathit{App}_1\,\frac{\Gamma : e \;\Downarrow\; \Delta[q \mapsto \lambda\,\boldsymbol{x}\,\boldsymbol{y}\,.\,e'] : q \qquad r \text{ fresh}}{\Gamma : e\,\boldsymbol{p} \;\Downarrow\; \Delta \cup [r \mapsto \lambda\,\boldsymbol{y}\,.\,e'[\boldsymbol{p}/\boldsymbol{x}]] : r}$$

$$\mathit{App}_2\,\frac{\Gamma : e \;\Downarrow\; \Delta[q \mapsto \lambda\,\boldsymbol{x}\,.\,e'] : q \qquad \Delta : e'[\boldsymbol{p}/\boldsymbol{x}]\,\boldsymbol{p'} \;\Downarrow\; \Theta : r}{\Gamma : e\,\boldsymbol{p}\,\boldsymbol{p'} \;\Downarrow\; \Theta : r}$$

$$\mathit{Let}\,\frac{\Gamma \cup [\boldsymbol{p} \mapsto \boldsymbol{lf}[\boldsymbol{p}/\boldsymbol{x}]] : e[\boldsymbol{p}/\boldsymbol{x}] \;\Downarrow\; \Delta : q \qquad \boldsymbol{p} \text{ fresh}}{\Gamma : \texttt{let}\ \boldsymbol{x} = \boldsymbol{lf}\ \texttt{in}\ e \;\Downarrow\; \Delta : q}$$

$$\mathit{Case}_1\,\frac{\Gamma : e \;\Downarrow\; \Delta[p \mapsto C_i\,\boldsymbol{p'}] : p \qquad \Delta : e_i[\boldsymbol{p'}/\boldsymbol{x_i}] \;\Downarrow\; \Theta : q}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \boldsymbol{C}\,\boldsymbol{x}\ \texttt{->}\ \boldsymbol{e} \;\Downarrow\; \Theta : q}$$

$$\mathit{Accu}\,\frac{}{\Gamma[p \mapsto k] : p \;\Downarrow\; \Gamma : p}$$

$$\mathit{App}_3\,\frac{\Gamma : e \;\Downarrow\; \Delta[q \mapsto k] : q \qquad \Delta \cup [r \mapsto \langle q\,p\rangle] : r\,\boldsymbol{p'} \;\Downarrow\; \Theta : s \qquad r \text{ fresh}}{\Gamma : e\,p\,\boldsymbol{p'} \;\Downarrow\; \Theta : s}$$

$$\mathit{Case}_2\,\frac{\Gamma : e \;\Downarrow\; \Delta[p \mapsto k] : p \qquad q \text{ fresh}}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \boldsymbol{alt} \;\Downarrow\; \Delta \cup [q \mapsto \langle \texttt{case}\ p\ \texttt{of}\ \boldsymbol{alt}\rangle] : q}$$

Figure 4: Semantics S4

$$Lam_{\uparrow}\ \frac{\Gamma \cup [q \mapsto \langle y\rangle] : p\, q \ \updownarrow\ \Delta : v \qquad q, y \text{ fresh}}{\Gamma[p \mapsto \lambda\, \boldsymbol{x} . e] : p \ \uparrow\ \Delta : \lambda\, y . v}$$

$$Cons_{\uparrow}\ \frac{\bigwedge_{i=1}^{|\boldsymbol{q}|} \quad \Gamma_i : q_i \ \updownarrow\ \Gamma_{i+1} : v_i}{\Gamma_1[p \mapsto C\,\boldsymbol{q}] : p \ \uparrow\ \Gamma_{|\boldsymbol{q}|+1} : C\,\boldsymbol{v}}$$

$$Accu_{\uparrow}\ \frac{\Gamma : k \ \uparrow_{\langle\rangle}\ \Delta : v}{\Gamma[p \mapsto k] : p \ \uparrow\ \Delta : v}$$

$$Var_{\uparrow_{\langle\rangle}}\ \frac{}{\Gamma : \langle x\rangle \ \uparrow_{\langle\rangle}\ \Gamma : x}$$

$$App_{\uparrow_{\langle\rangle}}\ \frac{\Gamma : k \ \uparrow_{\langle\rangle}\ \Delta : h\,\boldsymbol{v} \qquad \Delta : q \ \updownarrow\ \Theta : v'}{\Gamma[p \mapsto k] : \langle p\, q\rangle \ \uparrow_{\langle\rangle}\ \Theta : h\,\boldsymbol{v}\,v'}$$

$$Case_{\uparrow_{\langle\rangle}}\ \frac{\begin{array}{l}\Gamma : k \ \uparrow_{\langle\rangle}\ \Delta_1 : v \\ \bigwedge_{i=1}^{|\boldsymbol{alt}|} \quad \Delta_i \cup \Delta'_i : \texttt{case}\ p\ \texttt{of}\ \boldsymbol{alt} \ \updownarrow\ \Delta_{i+1} : v'_i \\ \quad \text{where } \boldsymbol{alt} = \boldsymbol{C}\,\boldsymbol{x}\ \texttt{->}\ \boldsymbol{e}, |\boldsymbol{q}| = |\boldsymbol{y}| = |\boldsymbol{x}| \text{ and } p, \boldsymbol{q}, \boldsymbol{y} \text{ fresh} \\ \qquad \text{and } \Delta'_i = [p \mapsto C_i\,\boldsymbol{q}] \cup [\boldsymbol{q} \mapsto \langle \boldsymbol{y}\rangle]\end{array}}{\Gamma[p \mapsto k] : \langle \texttt{case}\ p\ \texttt{of}\ \boldsymbol{alt}\rangle \ \uparrow_{\langle\rangle}\ \Delta_{|\boldsymbol{alt}|+1} : \texttt{case}\ v\ \texttt{of}\ \boldsymbol{C}\,\boldsymbol{y}\ \texttt{->}\ \boldsymbol{v'}}$$

Figure 5: Read Back Definition for S4

equivalent to S4. As usual, to keep track of subexpressions needed for later steps, we introduce a stack, containing function arguments, case alternatives and update frames $\#p$, marking heap closures for later thunk updates.

While most rules are standard, the rules dealing with accumulators are new. Accumulators act like other weak head normal forms by overwriting a thunk with an indirection when a update frame is present, cf. var_4. When a parameter is on the stack, app_3 grabs and saves it within a application accumulator. Similarly, $case_3$ grabs sequences of case alternatives.

The normalization using SSTG1 is defined by rule $Norm$ in figure 7. An expression e is evaluated in several steps to weak head normal form, as usual $\rightarrow^*$ denotes the transitive and reflexive closure of $\rightarrow$. We define $\mathcal{W}(\Delta : p : \boldsymbol{p'})$ to be true iff either p points to an constructor value or accumulator and $\boldsymbol{p'} = \Diamond$, or p points to a λ-abstraction with more formal parameters than present in $\boldsymbol{p'}$, thus detecting feasible final states of the transition system. This final state is then read back.

The read back definition for SSTG1 is given in figure 8. In contrast to the corresponding

	Heap	Control	Stack	Name
	$\Gamma \cup [p \mapsto e]$	p	S	var_1
$\rightarrow$	Γ	e	$\#p \cdot\!: S$	
	$\Gamma[p \mapsto \lambda\, \boldsymbol{x}\, \boldsymbol{y} \,.\, e]$	p	$\boldsymbol{p'} +\!\!+ \#q \cdot\!: S$	var_2
$\rightarrow$	$\Gamma \cup [q \Mapsto p\, \boldsymbol{p'}]$	p	$\boldsymbol{p'} +\!\!+ S$	
	$\Gamma[p \mapsto C\, \boldsymbol{p'}]$	p	$\#q \cdot\!: S$	var_3
$\rightarrow$	$\Gamma \cup [q \looparrowright p]$	p	S	
	$\Gamma[p \mapsto k]$	p	$\#q \cdot\!: S$	var_4
$\rightarrow$	$\Gamma \cup [q \looparrowright p]$	p	S	
	$\Gamma[p \looparrowright q]$	p	S	ind_1
$\rightarrow$	Γ	q	S	
	$\Gamma[p \Mapsto e]$	p	S	ind_2
$\rightarrow$	Γ	e	S	
	Γ	$e\, \boldsymbol{p}$	S	app_1
$\rightarrow$	Γ	e	$\boldsymbol{p} +\!\!+ S$	
	$\Gamma[p \mapsto \lambda\, \boldsymbol{x} \,.\, e]$	p	$\boldsymbol{p'} +\!\!+ S$	app_2
$\rightarrow$	Γ	$e[\boldsymbol{p'}/\boldsymbol{x}]$	S	
	$\Gamma[p \mapsto k]$	p	$p' \cdot\!: S$	app_3
$\rightarrow$	$\Gamma \cup [q \mapsto \langle p\, p' \rangle]$ where q fresh	q	S	
	Γ	$\texttt{let}\ \boldsymbol{x} = \boldsymbol{lf}\ \texttt{in}\ e$	S	let
$\rightarrow$	$\Gamma \cup [\boldsymbol{p} \mapsto \boldsymbol{lf}[\boldsymbol{p}/\boldsymbol{x}]]$ where $\boldsymbol{p}$ fresh	$e[\boldsymbol{p}/\boldsymbol{x}]$	S	
	Γ	$\texttt{case}\ e\ \texttt{of}\ \boldsymbol{alt}$	S	$case_1$
$\rightarrow$	Γ	e	$\boldsymbol{alt} \cdot\!: S$	
	$\Gamma[p \mapsto C_i\, \boldsymbol{p'}]$	p	$\boldsymbol{C}\, \boldsymbol{x}\ \texttt{->}\ \boldsymbol{e} \cdot\!: S$	$case_2$
$\rightarrow$	Γ	$e_i[\boldsymbol{p'}/\boldsymbol{x_i}]$	S	
	$\Gamma[p \mapsto k]$	p	$\boldsymbol{alt} \cdot\!: S$	$case_3$
$\rightarrow$	$\Gamma \cup [q \mapsto \langle \texttt{case}\ p\ \texttt{of}\ \boldsymbol{alt} \rangle]$ where q fresh	q	S	

Figure 6: Semantics SSTG1

$$Norm\ \frac{\Gamma : e : S \rightarrow^{*} \Delta : p : \boldsymbol{p'} \qquad \mathcal{W}(\Delta : p : \boldsymbol{p'}) \qquad \Delta : p : \boldsymbol{p'} \uparrow \Theta : v}{\Gamma : e : S \updownarrow \Theta : v}$$

Figure 7: Normalization using SSTG1

$$\mathit{Lam}_{\uparrow}\;\frac{\Gamma \cup [q \mapsto \langle y\rangle] : p : \boldsymbol{p'} \mathbin{:\!\cdot} q \;\updownarrow\; \Delta : v \qquad q, y \text{ fresh}}{\Gamma[p \mapsto \lambda\, \boldsymbol{x} \,.\, e] : p : \boldsymbol{p'} \;\uparrow\; \Delta : \lambda\, y \,.\, v}$$

$$\mathit{Cons}_{\uparrow}\;\frac{\bigwedge_{i=1}^{|\boldsymbol{q}|} \quad \Gamma_i : q_i : \Diamond \;\updownarrow\; \Gamma_{i+1} : v_i}{\Gamma_1[p \mapsto C\, \boldsymbol{q}] : p : \Diamond \;\uparrow\; \Gamma_{|\boldsymbol{q}|+1} : C\, \boldsymbol{v}}$$

$$\mathit{Accu}_{\uparrow}\;\frac{\Gamma : k \;\uparrow_{\langle\rangle}\; \Delta : v}{\Gamma[p \mapsto k] : p : \Diamond \;\uparrow\; \Delta : v}$$

$$\mathit{Var}_{\uparrow_{\langle\rangle}}\;\frac{}{\Gamma : \langle x\rangle \;\uparrow_{\langle\rangle}\; \Gamma : x}$$

$$\mathit{App}_{\uparrow_{\langle\rangle}}\;\frac{\Gamma : k \;\uparrow_{\langle\rangle}\; \Delta : h\, \boldsymbol{v} \qquad \Delta : q : \Diamond \;\updownarrow\; \Theta : v'}{\Gamma[p \mapsto k] : \langle p\, q\rangle \;\uparrow_{\langle\rangle}\; \Theta : h\, \boldsymbol{v}\, v'}$$

$$\mathit{Case}_{\uparrow_{\langle\rangle}}\;\frac{\begin{array}{l} \Gamma : k \;\uparrow_{\langle\rangle}\; \Delta_1 : v \\ \bigwedge_{i=1}^{|\boldsymbol{alt}|} \quad \Delta_i \cup \Delta'_i : p : \boldsymbol{alt} \mathbin{\cdot\!:} \Diamond \;\updownarrow\; \Delta_{i+1} : v'_i \\ \quad \text{where } \boldsymbol{alt} = \boldsymbol{C}\, \boldsymbol{x} \texttt{ -> } \boldsymbol{e}, |\boldsymbol{q}| = |\boldsymbol{y}| = |\boldsymbol{x}| \text{ and } p, \boldsymbol{q}, \boldsymbol{y} \text{ fresh} \\ \quad \text{and } \Delta'_i := [p \mapsto C_i\, \boldsymbol{q}] \cup [\boldsymbol{q} \mapsto \langle \boldsymbol{y}\rangle] \end{array}}{\Gamma[p \mapsto k] : \langle \texttt{case } p \texttt{ of } \boldsymbol{alt}\rangle \;\uparrow_{\langle\rangle}\; \Delta_{|\boldsymbol{alt}|+1} : \texttt{case } v \texttt{ of } \boldsymbol{C}\, \boldsymbol{y} \texttt{ -> } \boldsymbol{v'}}$$

Figure 8: Read Back Definition for SSTG1

definitions for S4, this time we avoid the generation of new code: while figure 5 created applications in rule $\mathit{Lam}_{\uparrow}$, the pseudo-argument is placed onto the stack in figure 8. Similarly rule $\mathit{Case}_{\uparrow_{\langle\rangle}}$ created case-expressions in figure 5 where we now place the case alternatives onto the stack, where they will be found by case_2 in the first evaluation step.

5 Environments instead of substitutions: SSTG2

As a step between the linearized semantics SSTG1 and the compiled of ISSTG in the next section, we defined SSTG2, not shown in this paper for space reasons. It replaces explicit substitutions with the modification of environments. The definition is fairly standard, especially since SSTG2 is close to STG2 of [dlEP03]. The modification includes the addition of an environment into the configuration of the abstract machine and to heap values possibly containing variables. Note that the rules dealing with accumulators do not perform substitutions, thus the translation of rules var_4, app_3 and $case_3$ is as simple as the translation of the rules for the read back relation.

To avoid memory leaks by storing unused variables in the environments, λ-forms and sequences of case alternative are annotated with *trimmers*, simple sequences of used variables, and environments are restricted to the variables contained in the trimmers when necessary. Annotated λ-forms and case branches look like $lf|_{\boldsymbol{x}}$ and $\boldsymbol{alt}|_{\boldsymbol{x}}$, respectively.

6 Imperative Code: ISSTG

The translation of FUN expressions to machine code is given in figure 9. We use a further restricted variant of FUN here, function positions in applications and scrutinees in case expressions are restricted to be mere variables, not arbitrary expressions. This simplifies the stack layout to enable the read back phase.

Since environments are split into stack parts for function arguments and closure parts allocated on the heap and accessed through a node pointer for other free variables of thunks, the translation functions use compile time environments for the stack and the node variables, denoted by ρ and η, respectively. We write variable lookups as $(\rho, \eta)\, x$, resulting in either $(\mathtt{STACK}, p)$ or $(\mathtt{NODE}, p)$, depending on the location of x on the stack or the heap. We require the domains of ρ and η to be disjoint. This lookup definition is complemented by the function ptr, used in the definition of the semantics of ISSTG, with $ptr\ (\mathtt{STACK}, i)$ being the i-th stack element and $ptr\ (\mathtt{NODE}, i)$ the i-th element of the closure environment.

The node environment η is a simple mapping from variable names to indices in the node variable list. The stack environment ρ is a tuple of a corresponding mapping from variable names to indices on the stack, and the number of local variables that have to be removed from the stack before entering another closure.

Additionally we use a code store cs and a branch information table bi. Both are modified by operations given after $\&$ as monadic side effects of the translation functions. The code store maps pointers to code sequences or branches, the latter mapping constructor names to code sequences. The branch information table is especially introduced for strong normalization and not present in [dlEP03]. It collects the number of variables saved on the stack for use by case branches, and records for which constructors branches exist and how many arguments these constructors have.

The definitions of the translation functions trE for expressions, $trAs$ and trA for case alternatives and trB for `let` bound λ-forms are best read together with the definition of the corresponding transition rules in figure 10 of the generated instructions.

The translation of applications builds a stack environment containing the function x and all arguments y, removes all local variables from the stack, and finally calls the function.

For case expressions, all variables needed in the branches and accessed via η are saved on the stack first. Then a pointer to the compiled alternatives and the scrutinee is left on the stack before the latter is entered. The notation ρ^{++} denotes a stack environment where all indices are incremented by one, thus taking care of the branch pointer pushed on the stack.

$$
\begin{array}{lcl}
\mathit{trE}\ (x\,\boldsymbol{y})\ \rho\ \eta & = & [\ \ \mathtt{BUILDENV}\ \ (\rho,\eta)\,x \cdot\!\!:\, (\rho,\eta)\,\boldsymbol{y}, \\
 & & \ \ \mathtt{SLIDE}\ \ (1+|\boldsymbol{y}|)\ \ (\mathit{snd}\ \rho), \\
 & & \ \ \mathtt{ENTER} \hspace{10em}]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trE}\ (\mathtt{case}\ x\ \mathtt{of}\ \boldsymbol{alt}|_{\boldsymbol{y}})\ \rho\ \eta & = & [\ \ \mathtt{BUILDENV}\ \ \boldsymbol{a}, \\
 & & \ \ \mathtt{PUSHALTS}\ \ p, \\
 & & \ \ \mathtt{BUILDENV}\ \ (\rho'^{+\!+},\eta)\,x, \\
 & & \ \ \mathtt{ENTER} \hspace{10em}] \\
 & \& & \mathit{bi} \cup [p \mapsto (\mathit{snd}\ \rho', \mathit{info}\ \boldsymbol{alt})] \\
\quad \text{where}\ \ \rho' & = & \rho + [y_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{y}|] \\
\quad p & = & \mathit{trAs}\ \boldsymbol{alt}\ \rho' \\
\quad \boldsymbol{a} & = & [(\mathtt{NODE},\eta\,z)\ \mid z \leftarrow \boldsymbol{y}\ \wedge\ z \in \mathit{dom}\ \eta] \\
\quad \mathit{info}\ (C\,\boldsymbol{x}\ \texttt{->}\ e) & = & (C,|\boldsymbol{x}|)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trE}\ (\mathtt{let}\ \boldsymbol{x} = \boldsymbol{lf}|_{\boldsymbol{y}}\ \mathtt{in}\ e)\ \rho\ \eta & = & [\ \ \mathtt{ALLOC}\ \ |\boldsymbol{y_i}|\ \ \mid i \leftarrow |\boldsymbol{y}|\,..1 \hspace{4em}]\ +\!\!+ \\
 & & [\ \ \mathtt{BUILDCLS}\ \ (i-1)\ \ p_i\ \ \boldsymbol{a_i}\ \ \mid i \leftarrow 1..|\boldsymbol{y}|\ \ ]\ +\!\!+ \\
 & & \ \ \mathit{trE}\ e\ \rho'\ \eta \\
\quad \text{where}\ \ \rho' & = & \rho + [x_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{x}|] \\
\quad \boldsymbol{p} & = & \mathit{trB}\ \boldsymbol{lf}|_{\boldsymbol{y}} \\
\quad \boldsymbol{a} & = & (\rho',\eta)\,\boldsymbol{y}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trAs}\ \boldsymbol{alt}\ \rho & = & p \\
 & \& & \mathit{cs} \cup [p \mapsto [\boldsymbol{C} \mapsto \mathit{trA}\ \boldsymbol{alt}\ \rho]] \\
\quad \text{where}\ \ \boldsymbol{alt} = \boldsymbol{C}\,\boldsymbol{x}\ \texttt{->}\ \boldsymbol{e} & & \\
\quad \text{and}\ \ p\ \text{fresh} & & \\
 & & \\
\mathit{trA}\ (C\,\boldsymbol{x}\ \texttt{->}\ e)\ \rho & = & \mathit{trE}\ e\ \rho\ [x_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{x}|]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{trB}\ (C\,\boldsymbol{x}\ |_{\boldsymbol{y}}) & = & p \\
 & \& & \mathit{cs} \cup [p \mapsto [\mathtt{RETURNCON}\ \ C]] \\
\quad \text{where}\ \ p\ \text{fresh} & & \\
 & & \\
\mathit{trB}\ (\lambda\,\boldsymbol{x}\,.\,e\ |_{\boldsymbol{y}}) & = & p \\
 & \& & \mathit{cs} \cup [p \mapsto [\mathtt{ARGCHECK}\ \ |\boldsymbol{x}|]\ +\!\!+\ \mathit{trE}\ e\ \rho\ \eta] \\
\quad \text{where}\ \ \rho & = & ([x_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{x}|], |\boldsymbol{x}|) \\
\quad \eta & = & [y_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{y}|] \\
\quad \text{and}\ \ p\ \text{fresh} & & \\
\mathit{trB}\ (e\ |_{\boldsymbol{y}}) & = & p \\
 & \& & \mathit{cs} \cup [p \mapsto [\mathtt{UPDMARK}]\ +\!\!+\ \mathit{trE}\ e\ \rho_{\emptyset}\ \eta] \\
\quad \text{where}\ \ \rho_{\emptyset} & = & (\emptyset, 0) \\
\quad \eta & = & [y_i \mapsto i-1\ \mid i \leftarrow 1..|\boldsymbol{y}|] \\
\quad \text{and}\ \ p\ \text{fresh} & &
\end{array}
$$

Figure 9: Translation of FUN to Machine Code

	Code	Stack	Node	Heap
	$[\texttt{ENTER}]$	$q \cdot\!: S$	p	$\Gamma[q \mapsto (r, \boldsymbol{r'})]$
$\rightarrow$	is	S	q	Γ
	where $cs[r \mapsto is]$			
	$[\texttt{RETURNCON}\ C]$	$q \cdot\!: S$	p	Γ
$\rightarrow$	is	S	p	Γ
	where $cs[q \mapsto bt]$ and $bt[C \mapsto is]$			
	$[\texttt{RETURNCON}\ C]$	$\#q \cdot\!: S$	p	Γ
$\rightarrow$	$[\texttt{RETURNCON}\ C]$	S	p	$\Gamma \cup [q \mapsto (p_{ind}, p)]$
	$\texttt{ARGCHECK}\ n \cdot\!: is$	$\boldsymbol{q} +\!\!+ S$	p	Γ
$\rightarrow$	is	$\boldsymbol{q} \cdot\!: S$	p	Γ
	where $n = \lvert\boldsymbol{q}\rvert$			
	$\texttt{ARGCHECK}\ n \cdot\!: is$	$\boldsymbol{q} +\!\!+ \#r \cdot\!: S$	p	$\Gamma \cup [r \mapsto (p_{bh}, \boldsymbol{q'})]$
$\rightarrow$	$\texttt{ARGCHECK}\ n \cdot\!: is$	$\boldsymbol{q} +\!\!+ S$	p	$\Gamma \cup [r \mapsto (p_{pap}, p \cdot\!: \boldsymbol{q})]$
	where $n > \lvert\boldsymbol{q}\rvert$			
	$\texttt{ALLOC}\ n \cdot\!: is$	S	p	Γ
$\rightarrow$	is	$q \cdot\!: S$	p	Γ'
	where q points to a new closure with space for n variables and Γ' is the resulting heap after allocation			
	$\texttt{BUILDCLS}\ i\ q\ \boldsymbol{a} \cdot\!: is$	S	p	Γ
$\rightarrow$	is	S	p	$\Gamma \cup [S!i \mapsto (q, ptr\ \boldsymbol{a})]$
	$\texttt{BUILDENV}\ \boldsymbol{a} \cdot\!: is$	S	p	Γ
$\rightarrow$	is	$ptr\ \boldsymbol{a} +\!\!+ S$	p	Γ
	$\texttt{PUSHALTS}\ q \cdot\!: is$	S	p	Γ
$\rightarrow$	is	$q \cdot\!: S$	p	Γ
	$\texttt{UPDMARK} \cdot\!: is$	S	p	$\Gamma \cup [p \mapsto (q, \boldsymbol{q'})]$
$\rightarrow$	is	$\#p \cdot\!: S$	p	$\Gamma \cup [p \mapsto (p_{bh}, \boldsymbol{q'})]$
	$\texttt{SLIDE}\ n\ m \cdot\!: is$	$\boldsymbol{q} +\!\!+ \boldsymbol{q'} +\!\!+ S$	p	Γ
$\rightarrow$	is	$\boldsymbol{q} +\!\!+ S$	p	Γ
	where $n = \lvert\boldsymbol{q}\rvert$ and $m = \lvert\boldsymbol{q'}\rvert$			
	$\texttt{PAP} \cdot\!: is$	S	p	$\Gamma[p \mapsto (q, \boldsymbol{q'})]$
$\rightarrow$	is	$\boldsymbol{q'} +\!\!+ S$	p	Γ
	$[\texttt{ACCU}]$	$q \cdot\!: S$	p	$\Gamma[p \mapsto (p_{accu}, k)]$
$\rightarrow$	$[\texttt{ACCU}]$	S	r	$\Gamma \cup [r \mapsto (p_{accu}, \langle p\, q\rangle)]$
	where r fresh			
	$[\texttt{ACCU}]$	$\#q \cdot\!: S$	p	$\Gamma \cup [q \mapsto (p_{bh}, \boldsymbol{q'})]$
$\rightarrow$	$[\texttt{ACCU}]$	S	p	$\Gamma \cup [q \mapsto (p_{ind}, p)]$
	$[\texttt{ACCU}]$	$q \cdot\!: \boldsymbol{q'} +\!\!+ S$	p	$\Gamma[p \mapsto (p_{accu}, k)]$
$\rightarrow$	$[\texttt{ACCU}]$	S	r	$\Gamma \cup [r \mapsto (p_{accu}, k')]$
	where $bi[q \mapsto (n, info)]$, $\lvert\boldsymbol{q'}\rvert = n$, $k' = \langle \texttt{case}\ p\ \texttt{of}\ (q, \boldsymbol{q'})\rangle$ and r fresh,			

Figure 10: Semantics ISSTG

$$Norm\frac{\begin{array}{ll}[\mathtt{ENTER}] : S : q : \Gamma \rightarrow^{*} is : \boldsymbol{p'} : p : \Delta & \\ \mathcal{W}(is : \boldsymbol{p'} : p : \Delta) & \Delta : p : \boldsymbol{p'} \uparrow \Theta : v\end{array}}{\Gamma : S \updownarrow \Theta : v}$$

Figure 11: Normalization using ISSTG

Local bindings are compiled to a sequence of ALLOC instructions to reserve heap space, that is in turn filled by a sequence of BUILDCLS instructions, before the compiled body is executed. The bound λ-forms are compiled by trB. Constructors are compiled to a single instruction, RETURNCON, that dereferences the branch pointer found on the stack and selects the right branch. Compiled abstractions check whether enough arguments are present on the stack via ARGCHECK. When this check is positive, the body of the abstraction is executed, otherwise an update frame is expected on the stack and the corresponding closure on the heap is overwritten by a partial application (*pap*). The code of other bound expressions starts with UPDMARK that pushes an update frame.

During execution we expect several code sequences preallocated in the code store. For indirections, we make use of p_{ind} with $cs[p_{ind} \mapsto [\mathtt{BUILDENV}\ (\mathtt{NODE}, 0), \mathtt{ENTER}]]$. For blackholing, that is for marking expressions under evaluation, we use p_{bh} pointing to an empty code sequence, i. e. $cs[p_{bh} \mapsto \Diamond]$, such that evaluation gets stuck when an expression is re-entered before reaching WHNF. When allocating partial applications, we use p_{pap} with $cs[p_{pap} \mapsto [\mathtt{PAP}, \mathtt{ENTER}]]$. And finally, for accumulator allocations, we use p_{accu} with $cs[p_{accu} \mapsto [\mathtt{ACCU}]]$, i. e. p_{accu} points to the code that grabs either function arguments or case alternatives together with their environments from the stack, using the branch information table where necessary.

The normalization rule, given in figure 11, now normalizes a *stack*. The topmost stack element is entered, taking the remaining stack values as function arguments or pointers to case branches. The necessary adoptions for read back are shown in figure 12.

7 Conclusion and Related Work

We used the derivation of the STG machine presented in [dlEP03] as a basis for our strong normalization system. At this, we used the idea of accumulators and a separate read back phase as presented in [GL02], adopting them for lazy evaluation and the STG machine.

Correctness proofs are ongoing, proving the equivalence of normalization with different semantics, i.e. $\exists\Delta.\emptyset : e \updownarrow^{s_i} \Delta : v \Leftrightarrow \exists\Delta'.\emptyset : e \updownarrow^{s_{i+1}} \Delta' : v$ with the s_i from {S4, SSTG1, SSTG2, ISSTG}. The steps between S4, SSTG1 and SSTG2 are already completed. We expect no difficulties for the step to ISSTG. Upon completion, we intend the publication of the proofs as a technical report.

In [Cré90] an abstract machine is presented for strong normalization of λ terms. It differs from ours and Grégoire's in the missing distinction of weak evaluation and read back. This

$$\mathit{Lam}_{\uparrow}\ \frac{\Gamma \cup [q \mapsto (p_{accu}, \langle y \rangle)] : p \cdot: \boldsymbol{p'} :\cdot\, q \;\updownarrow\; \Delta : v \qquad \begin{array}{l} q, y \text{ fresh} \\ cs[r \mapsto \texttt{ARGCHECK} \cdot: is] \end{array}}{\Gamma[p \mapsto (r, \boldsymbol{r'})] : p : \boldsymbol{p'} \;\uparrow\; \Delta : \lambda\, y \,.\, v}$$

$$\mathit{Cons}_{\uparrow}\ \frac{\bigwedge_{i=1}^{|\boldsymbol{q}|} \quad \Gamma_i : [q_i] \;\updownarrow\; \Gamma_{i+1} : v_i \qquad cs[r \mapsto [\texttt{RETURNCON}\ C]]}{\Gamma_1[p \mapsto (r, \boldsymbol{r'})] : p : \Diamond \;\uparrow\; \Gamma_{|\boldsymbol{q}|+1} : C\, \boldsymbol{v}}$$

$$\mathit{Accu}_{\uparrow}\ \frac{\Gamma : k \;\uparrow_{\langle\rangle}\; \Delta : v \qquad cs[r \mapsto [\texttt{ACCU}]]}{\Gamma[p \mapsto (r, k)] : p : \Diamond \;\uparrow\; \Delta : v}$$

$$\mathit{Var}_{\uparrow_{\langle\rangle}}\ \frac{}{\Gamma : \langle x \rangle \;\uparrow_{\langle\rangle}\; \Gamma : x}$$

$$\mathit{App}_{\uparrow_{\langle\rangle}}\ \frac{\Gamma : k \;\uparrow_{\langle\rangle}\; \Delta : h\, \boldsymbol{v} \qquad \Delta : [q] \;\updownarrow\; \Theta : v'}{\Gamma[p \mapsto (r, k)] : \langle p\, q \rangle \;\uparrow_{\langle\rangle}\; \Theta : h\, \boldsymbol{v}\, v'}$$

$$\mathit{Case}_{\uparrow_{\langle\rangle}}\ \frac{\begin{array}{l} \Gamma : k \;\uparrow_{\langle\rangle}\; \Delta_1 : v \\ \bigwedge_{i=1}^{|\boldsymbol{C}|} \quad \Delta_i \cup \Delta'_i : p' \cdot: q \cdot: q' \;\updownarrow\; \Delta_{i+1} : v'_i \\ \quad \text{where } bi[q \mapsto (n, (\boldsymbol{C}, \boldsymbol{m}))], |\boldsymbol{p''}| = |\boldsymbol{y}| = m_i \text{ and } p', \boldsymbol{p''}, \boldsymbol{y} \text{ fresh} \\ \quad \text{and } \Delta'_i = [p' \mapsto (p_{C_i}\, \boldsymbol{p''})] \cup [\boldsymbol{p''} \mapsto (p_{accu}, \langle \boldsymbol{y} \rangle)] \end{array}}{\Gamma[p \mapsto (r, k)] : \langle \texttt{case}\ p\ \texttt{of}\ (q, \boldsymbol{q'}) \rangle \;\uparrow_{\langle\rangle}\; \Delta_{|\boldsymbol{alt}|+1} : \texttt{case}\ v\ \texttt{of}\ \boldsymbol{C}\, \boldsymbol{y}\ \texttt{->}\ \boldsymbol{v'}}$$

Figure 12: Read Back Definition for ISSTG

might be faster when reducing to normal forms, but when the normalization system is used for dependent type systems, it prevents some improvements as detailed in [GL02, Kle08].

In [Klu04, chapter 10] a lazy variant of the π-RED system is presented, that can strongly reduce functional programs. Its η-extension mechanism corresponds to our rule $Lam_{\uparrow}$. The π-RED system is based on a tagging mechanism instead of the STG machine, thus we had to find a way to implement free variables without using tag bits, resulting in accumulators with the same calling conventions as ordinary functions and constructors in STG code. Additionally, the focus of π-RED is broader: the system allows to reconstruct source code for unevaluated closures. This broader focus complicates matters, in π-RED the machine instructions have several modes depending on a reduction counter, and function pointers are referenced indirectly through descriptors. Since we are only interested in weak and strong normal forms, the ISSTG design keeps simpler.

Our approach is related to untyped normalization by evaluation (nbe) [AJ04], where λ-

terms are translated to their semantical meaning, from which normal forms are extracted. In our case, the meaning of a FUN expression can be seen as the state transitions of the running machine code, from which the normal forms are obtained by read back. But our approach implements e.g. function calls more efficient: A straightforward definition of untyped nbe as given in [AJ04] has to check at each application whether the applied function is semantical or syntactical, while in ISSTG the applied function can be called directly because of the calling convention of accumulators.

Another related line of research is partial evaluation, most closely type-directed partial evaluation (tdpe) introduced by Danvy in [Dan96]. This evaluator generates constructors expressions not only when free variables are scrutinized in case expressions, but whenever functions take arguments of disjoint sum types, introducing additional case analyses. These analyses change the strictness properties of normalized expressions, even non-strict functions are normalized to strict functions. This is avoided by our approach of residuating case expressions only when free variables are analyzed by a case expression. For the same reason our implementation deals better with recursive data types: tdpe cannot create pseudo arguments for functions with inductively defined domain types. Moreover, tdpe produces η-long normal forms, which is not feasible in some dependently typed systems.

References

[AJ04] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalisation by evaluation. *Mathematical Structures in Computer Science*, 14(04):587–611, 2004.

[Cré90] P. Crégut. An abstract machine for Lambda-terms normalization. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 333–340. ACM Press, 1990.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–257. ACM Press, 1996.

[dlEP03] Alberto de la Encina and Ricardo Peña. Formally deriving an STG machine. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on principles and practice of declaritive programming*, pages 102–112. ACM Press, 2003.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 235–246. ACM Press, 2002.

[Kle08] Dirk Kleeblatt. Checking dependent types using compiled code. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2008.

[Klu04] Werner Kluge. *Abstract computing machines*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[PJ92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992.