# Formalizing Resource Allocation in a Compiler

Peter Thiemann

Department of Computer Science
University of Nottingham
Nottingham NG7 2RD, England
pjt@cs.nott.ac.uk

**Abstract.** On the basis of an A-normal form intermediate language we formally specify resource allocation in a compiler for a strict functional language. Here, resource is to be understood in the most general sense: registers, temporaries, data representations, etc. All these should be (and can be, but have never been) specified formally. Our approach employs a non-standard annotated type system for the formalization. Although A-normal form turns out not to be the ideal vehicle for this investigation, we can prove some basic properties using the formalization.

## 1 Resource Allocation

Resource allocation in the back end of a compiler is often poorly specified. More often than not register allocation, administration of temporaries, and representation conversions are only specified procedurally [1, 6]. Code generators based on such algorithmic specifications can be hard to maintain or prove correct. Even the authors of such code generators are sometimes not aware of all the invariants that must be preserved.

Therefore, we investigate a declarative approach to resource allocation in the back end of a compiler. The approach is based on an annotated type system of *implementation types* that makes resource allocation and conversion explicit. The use of type conversion rules enables us to defer memory and register allocation until the context of use forces an allocation. For example, a constant initially leads to an annotation of the type of the variable holding the constant without generating any code. The annotation holds the "immediate value" of the constant. There are type conversion rules that change the annotation from "immediate value" to "value in register $k$" and generate a corresponding piece of code if the context of use requires the value of the variable in a register. Further conversion rules create or remove indirection. The indirection rules move a value to memory and change the annotation to "value in memory at address $R[k]+i$", where $R[k]$ is an address in register $k$ and $i$ is an offset. The indirection removing rules work the other way round. The indirection rules usually apply to the arguments of function calls or to values that are put into data structures. Spilling the contents of registers is another application of the last kind of conversion rules. Other rules may make direct use of immediate values, for example when generating instructions with immediate operands.

The resulting high degree of flexibility allows for arbitrary intra-module calling conventions. Since the calling convention is part of every function's type, each function "negotiates" its convention with all its call sites. Contrast this with the algorithm used in the SML/NJ compiler [2] where the first call encountered by the code generator determines the calling convention for a procedure. Obviously, this is one pragmatic way of negotiating, but surely not a declarative one (nor a democratic one). External functions can have arbitrary calling conventions, too, as long as their implementation type is known. If the external functions are unknown, any standard calling convention (including caller saves/callee saves registers) can be enforced just by imposing a suitable implementation type. The same holds for exported functions, where the only requirement is that their implementation type is also exported, for example, in an interface file.

Implementation types can also model some other worthwhile optimizations. For example, a *lightweight closure* does not contain all free variables of a function. It can only be used correctly if all variables that it is not closed over are available at all call sites. Implementation types can guarantee the correctness of a variant of lightweight closure conversion (cf. [20]). In our case, this conversion does not take place at the level of the source language, it rather happens while translating to actual machine code. The translation ensures that the values that are not put into the closure are available at all call sites.

### 1.1 Overview

In the next section, we define the source language, its operational semantics, the implementation type language, and the target language. The introduction and discussion of the typing rules is subject of Section 3. Section 4 documents some properties of the system. Finally, we discuss related work (Sec. 5) and draw conclusions (Sec. 6).

## 2 Language

We have chosen a simply typed lambda calculus in A-normal form, a typical intermediate language used in compilers, as the starting point of our investigation. Compiling with A-normal forms [5] is said to yield the principal benefits of compiling with continuations (explicit control flow, naming of intermediate results, making continuations explicit) without incurring the overhead of actually transforming the program to continuation-passing style and without complicating the types in the intermediate language.

### 2.1 Terms

We strengthen our requirements somewhat with respect to the usual definition of A-normal form. Figure 1 defines the terms of *restricted A-normal form.* There are computation terms $a$ and value terms $v$. Value terms are integer constants

$$a ::= \mathsf{let}\ x = v\ \mathsf{in}\ a \mid \mathsf{let}\ x = x\ @\ x\ \mathsf{in}\ a \mid \mathsf{let}\ x = x + x\ \mathsf{in}\ a \mid$$
$$x \mid x\ @\ x$$
$$v ::= n \mid x \mid \lambda x.a$$

**Fig. 1.** Restricted A-normal form: terms

$n$, variables $x$, or lambda abstractions $\lambda x.a$. Computation terms either sequentialize computations $\mathsf{let}\ x = \ldots\ \mathsf{in}\ a$ or they return a result, which can either be the value of a variable or it can take the form of a tail call to some function. Usually, A-normal form [5] only requires the arguments of applications and primitives to be values $v$. Restricted A-normal form requires variables $x$ in all these places. With this restriction, no resource allocation occurs "inside" of a term and resource conversions can be restricted to occur between some $\mathsf{let}$ and its body, without lack of generality.

### 2.2 Operational semantics

The semantics is defined by a fairly conventional CEK machine (see Fig. 2). A machine state is a triple $(a, \rho, \kappa)$ where

- $a \in \mathsf{Term}$ is a term in A-normal form,
- $\rho \in \mathsf{Env} = \mathsf{Var} \rightharpoonup \mathsf{Val}$ is an environment, and
- $\kappa \in \mathsf{K}$ is a continuation where $\mathsf{K} = \mathsf{Void} + \mathsf{Env} \times \mathsf{Var} \times \mathsf{Term} \times \mathsf{K}$.

Here, partial functions are denoted by $\rightharpoonup$, $\rho|_F$ restricts the domain of $\rho$ to $F$, $\mathsf{Void}$ is a one-element set, and $+$ denotes disjoint union of sets. A value $\in \mathsf{Val}$ is either $\mathsf{Num}\ (n)$ or $\mathsf{Fun}\ (\rho, \lambda y.a)$ where $\rho \in \mathsf{Env}$ and $\lambda y.a \in \mathsf{Term}$.

Inspection of the last rule reveals that the semantics enforces proper tail recursion, because the function call in tail position does not create a new continuation. The transitions that state additional constraints are undefined if the constraints are not met.

### 2.3 Types

Figure 3 defines implementation types, which form an extension of simple types. In an implementation type, each type constructor carries a location $l$. If the location is $\varepsilon$ ("not allocated") then the information corresponding to the type constructor is only present in the type. For example, (going beyond the fragment considered in this paper) if the product type constructor $\times$ carries the annotation $\varepsilon$ then only its components are allocated as prescribed by their locations; the pair itself is not physically represented.

If the location is $\mathsf{imm}\ n$ ("immediate value $n$") then the value corresponding to the type carrying this location is know to be the integer $n$. The name comes from the immediate addressing mode that is present in many architectures, and

$$
\begin{array}{ll}
(\mathsf{let}\ x = n\ \mathsf{in}\ a, \rho, \kappa) & \mapsto (a, \rho[x \mapsto \mathsf{Num}\ (n)], \kappa) \\
(\mathsf{let}\ x = y\ \mathsf{in}\ a, \rho, \kappa) & \mapsto (a, \rho[x \mapsto \rho(y)], \kappa) \\
(\mathsf{let}\ x = \lambda y.a'\ \mathsf{in}\ a, \rho, \kappa) & \mapsto (a, \rho[x \mapsto \mathsf{Fun}\ (\rho|_{FV(\lambda y.a')}, \lambda y.a')], \kappa) \\
(\mathsf{let}\ x = w\ @\ z\ \mathsf{in}\ a, \rho, \kappa) & \mapsto (a', \rho'[y \mapsto \rho(z)], \langle \rho|_{FV(\mathsf{let}\ x=w\ @\ z\ \mathsf{in}\ a)}, x, a, \kappa\rangle) \\
& \quad \text{if } \rho(w) = \mathsf{Fun}\ (\rho', \lambda y.a') \\
(\mathsf{let}\ x = w + z\ \mathsf{in}\ a, \rho, \kappa) & \mapsto (a, \rho[x \mapsto \mathsf{Num}\ (m+n)], \kappa) \\
& \quad \text{if } \rho(w) = \mathsf{Num}\ (m) \text{ and } \rho(z) = \mathsf{Num}\ (n) \\
(z, \rho', \langle \rho, x, a, \kappa\rangle) & \mapsto (a, \rho[x \mapsto \rho'(z)], \kappa) \\
(w + z, \rho', \langle \rho, x, a, \kappa\rangle) & \mapsto (a, \rho[x \mapsto \mathsf{Num}\ (m+n)], \kappa) \\
& \quad \text{if } \rho(w) = \mathsf{Num}\ (m) \text{ and } \rho(z) = \mathsf{Num}\ (n) \\
(w\ @\ z, \rho, \kappa) & \mapsto (a', \rho'[y \mapsto \rho(z)], \kappa) \\
& \quad \text{if } \rho(w) = \mathsf{Fun}\ (\rho', \lambda y.a')
\end{array}
$$

**Fig. 2.** Operational semantics

$$
\begin{array}{lcl}
\tau & ::= & (\sigma; l) \\
\sigma & ::= & \mathsf{int} \mid \tau \xrightarrow{F,P,M,F',k} \mathsf{cont}\ l\ \tau \mid \mathsf{cont}\ F \\
l & ::= & \varepsilon \mid \mathsf{imm}\ n \mid A[\mathsf{reg}\ n] \\
A & ::= & [\,] \mid \mathsf{mem}\langle i, A\rangle
\end{array}
$$

**Fig. 3.** Syntax of implementation types

immediate values are expected to take part in generating instructions using immediate addressing.

If the location is $\mathsf{reg}\ k$ then the value of that type is resident in register $k$. In addition, the register might hold an indirection, i.e., the address of a block of memory where the value is stored at some offset $i$: $\mathsf{mem}\langle i, \mathsf{reg}\ k\rangle$. In general, this indirection step may be repeated an arbitrary number of times, which is expressed by $\mathsf{mem}\langle i, A[\mathsf{reg}\ k]\rangle$.

There are two syntactic categories for types. $\tau$ ranges over implementation types, i.e., $\tau$ is a pair of a "stripped" implementation type $\sigma$ and the location of its top-level type constructor. For this paper, $\sigma$ ranges over $\mathsf{int}$, the type of integers, and $\tau_2 \xrightarrow{F,P,M,F',k} \mathsf{cont}\ l\ \tau_1$, the type of functions that map objects of type $\tau_2$ to objects of type $\tau_1$ involving a continuation closure at location $l$, and $\mathsf{cont}\ F$ , the type of a continuation. The annotation $F, P, M, F', k$ on the function arrow is reminiscent to effect systems [7, 11]. It determines the latent resource usage of the function, which becomes effective when the function is called. It is explained in the next section 3 together with the judgements of implementation typing. The last alternative, $\sigma \equiv \mathsf{cont}\ F$  is the type of a continuation identifier. This type carries the location information of the current continuation, which would otherwise be lost (see Sec. 3).

### 2.4 Additional conventions

The architecture of a real processor places certain limits on the use of registers. For example, processors may have

- dedicated floating-point registers;
- special address registers ("pointers" to closures and tuples);
- special register(s) for continuations;
- special register(s) for condition codes.

In addition, the number of such registers is limited. These restrictions are modeled by a function Regs

$$\mathsf{Regs} : \mathsf{TypeConstructor} \rightarrow \mathcal{P}(\mathsf{RegisterNames})$$

that maps a type constructor to a set of register names (which might be represented by integers). Occasionally, we apply Regs to a stripped implementation type $\sigma$ when it should be applied to the top-level type constructor of $\sigma$. We do not define Regs here since it depends on the particular architecture that we want to generate code for.

### 2.5 Target code

The target code of the translation is an assembly language for an abstract RISC processor. It has the following commands, expressed in a rather suggestive way with $R[k]$ denoting register reference and $M[a]$ denoting a memory reference. Here, $k, j \in \mathsf{RegisterNames}$, $a, i$ are memory addresses for data, $t$ is a symbolic label for a code address, and $n$ is an integer.

| | | |
|---|---|---|
| $t :$ | | label declaration |
| $R[k]$ | $:= n$ | load numeric constant |
| $R[k]$ | $:= t$ | load address constant |
| $R[k]$ | $:= R[i] + R[j]$ | arithmetic operation |
| $R[k]$ | $:= n + R[j]$ | arithmetic operation |
| $R[k]$ | $:= M[i + R[j]]$ | load indirect with offset |
| $M[i + R[j]]$ | $:= R[k]$ | store indirect with offset |
| $R[k]$ | $:= \mathsf{Allocate}(n)$ | memory allocation |
| $\mathsf{Goto}\ t$ | | unconditional jump |
| $\mathsf{Goto}\ R[i]$ | | unconditional indirect jump |

The infix operator ";" performs concatenation of code sequences. We identify singleton code sequences with single instructions. For simplicity, we assume that all data objects have a standard representation of the same size (which might be a pointer).

The state of the abstract processor is a triple $\langle C, \overline{R}, \overline{M} \rangle$ where $C$ is a code sequence, $\overline{R}$ is the register bank (a mapping from a finite set of register names to data), and $\overline{M}$ is the memory (a mapping from an infinite set of addresses to data). The program store, which maps labels (code addresses) to code sequences, is left implicit. The instruction $\mathsf{Allocate}(n)$ returns the address of a contiguous block of memory of size $n$. It guarantees that there is no overlap with previously allocated blocks, i.e., it never returns the same address twice. Some of our proofs exploit this guarantee by relying on the uniqueness of data addresses for identification. In practice there will be a garbage collector that maps the infinite address space into a finite one, which removes old unreachable addresses from the system.

## 3 Typing

The typing judgement is reminiscent to that of an effect system [7, 11]. The typing process determines a translation to abstract assembly code as defined above. Therefore, we use a translation judgement $\Gamma, P, F, S \vdash a : \tau; M, F', C$ to describe both together. In every judgement,

- $\Gamma$ is a type assumption, i.e., a list of pairs $x : \tau$. By convention, type assumptions are extended by appending a new pair on the right as in $\Gamma, x : \tau$. The same notation $\Gamma, x : \tau$ also serves to extract the rightmost pair from a type assumption.
- $P$ is a set of *preserved registers*. The translation guarantees that all registers $k \in P$ hold the same value after evaluation of $a$ as before, but during evaluation of $a$ these values may be spilled and register $k$ may hold different values temporarily. Members of $P$ correspond to callee-saves registers.
- $F, F'$ are sets of *fixed registers*. The translation guarantees that a register $k \in F$ is not used as long as there is some reference to it in the type assumption or in the context modeled by $F'$. Furthermore, it expects that the context of $a$ handles the registers mentioned in $F'$ in the same way. Members of $F$ must not be spilled. However, if there is no reference remaining to some $k \in F$ then $k$ may be removed from $F$.
  The main use of $F$ and $F'$ is lightweight closure conversion and avoiding the allocation of closures altogether. In both cases, the type assumption contains a variable $w$ of type $\tau_2 \xrightarrow{F_1, P_1, M_1, F_1', k_1} \mathsf{cont}\ l\ \tau_1$ where $F_1$ describes the components of the closure that have not been allocated (i.e., they reside in registers drawn from $F_1$). Consequently, $F_1 \subseteq F$ must hold at a call site $a \equiv \mathsf{let}\ x = w\ @\ z\ \mathsf{in}\ a'$ so that all registers in $F_1$ contain the correct values.
- $S$ is a list of reloads of the form $\langle k, i_1 \dots i_p \rangle \dots$ where register $k$ points to a spill area and $i_1$ through $i_p$ are the spilled registers. The notation $\varepsilon$ is used for the empty list of reloads, i.e., when all values are either implicit or reside in registers: $S = \varepsilon$ means that nothing is currently spilled.
- $M$ is a set of registers that are possibly modified while evaluating $a$.
- $C$ is code of the target machine (see Sec. 2.5).

Before we start discussing the typing rules proper, we need to define the set of registers referenced from a type assumption.

**Definition 1.** *The* reference set *of a location, type, or type assumption is the set of registers that the location, type, or type assumption refers to.*

- $\mathsf{Refer}\ \varepsilon = \emptyset$, $\mathsf{Refer}\ (\mathsf{imm}\ n) = \emptyset$, $\mathsf{Refer}\ (A[\mathsf{reg}\ n]) = \{n\}$;
- $\mathsf{Refer}\ (\mathsf{int}; l) = \mathsf{Refer}\ l$;
- $\mathsf{Refer}\ (\tau_2 \xrightarrow{F,P,M,F',k} \mathsf{cont}\ l'\ \tau_1; l) = \mathsf{Refer}\ l \cup F$;
- $\mathsf{Refer}\ (\mathsf{cont}\ F\ ; l) = \mathsf{Refer}\ l \cup F$;
- $\mathsf{Refer}\ \Gamma = \bigcup_{x:\tau\in\Gamma} \mathsf{Refer}\ \tau$.

### 3.1 Typing rules

The typing rules are organized into context rules that manipulate type assumptions, value rules that provide typings for variables and constants, representation conversion rules, computation rules that deal with let expressions, and return rules that describe returning values from function invocations. Of these rules, the context rules and the conversion rules are nondeterministic, the remaining rules are tied to specific syntactic constructs, i.e., syntax-directed.

**Context rules** Each use of a variable consumes an element of the type assumption. This convention saves us from spilling dead variables since a "good" derivation only duplicates variables that are still live. Hence there is a rule to duplicate assumptions $x : \tau$.

$$\textit{(dup)}\ \frac{(\Gamma, x : \tau, x : \tau), P, F, S \vdash a : \tau', M, F', C}{(\Gamma, x : \tau), P, F, S \vdash a : \tau', M, F', C}$$

There is a dual weakening rule that drops a variable assumption. The set of fixed registers is updated accordingly. Dropping of variable assumptions starts on the left side of a type assumption to avoid problems with shadowed assumptions.

$$\textit{(weak)}\ \frac{\Gamma, P, F \cap \mathsf{Refer}\ \Gamma, S \vdash a : \tau', M, F', C}{(x : \tau, \Gamma), P, F, S \vdash a : \tau', M, F', C}$$

Finally, there is a rule to organize access to the type assumptions. It exchanges adjacent elements of the type assumption *provided that they bind different variables.*

$$\textit{(exch)}\ \frac{\Gamma, y : \tau_2, x : \tau_1, \Gamma', P, F, S \vdash a : \tau', M, F', C}{\Gamma, x : \tau_1, y : \tau_2, \Gamma', P, F, S \vdash a : \tau', M, F', C}\quad x \not\equiv y$$

The explicit presence of these rules is reminiscent of linear type systems [8, 23].

**Value rules** And here is a simple rule that consumes a variable assumption for $y : \tau$ at the price of producing one for $x : \tau$.

$$\textit{(let-var)}\ \frac{(\Gamma, x : \tau), P, F, S \vdash a : \tau'; M, F', C}{(\Gamma, y : \tau), P, F, S \vdash \mathsf{let}\ x = y\ \mathsf{in}\ a : \tau'; M, F', C}$$

Application of the *(let-var)* rule does not imply a change in the actual location of the value. The variable $x$ becomes an alias for $y$ in the expression $a$. The rule can be eliminated in favor of a reduction rule for expressions in restricted A-normal form: $\mathsf{let}\ x = y\ \mathsf{in}\ a \mapsto a[x := y]$ (capture-avoiding substitution of $y$ for $x$ in $a$). There is no penalty for this reduction, because the system allows the conversion of each occurrence of a variable individually. So former occurrences of $x$ can still be treated differently than former occurrences of $y$.

A constant starts its life as an immediate value which is only present in the implementation type. The typing derivation propagates this type and value to the point where it either selects an instruction with an immediate operand or where the context forces allocation into a register.

$$\textit{(let-const)}\ \frac{(\Gamma, x : (\mathsf{int}; \mathsf{imm}\ n)), P, F, S \vdash a : \tau'; M, F', C}{\Gamma, P, F, S \vdash \mathsf{let}\ x = n\ \mathsf{in}\ a : \tau'; M, F', C}$$

**Conversion rules** Some primitives expect their arguments allocated in registers. As we have seen, values are usually not born into registers. So, how do they get there? The solution lies in conversion rules that transform the type assumption. These rules generate code and allocate registers.

A register $k$ is deemed available if it is neither referred to by $\Gamma$ nor mentioned in $P \cup F$: $k \notin \mathsf{Refer}\ \Gamma \cup P \cup F$.

Immediate integer values generate a simple load instruction. In this case, the register selected must be suitable for an integer ($k \in \mathsf{RegisterNamesint}$) besides being available for allocation.

$$\textit{(conv-imm)}\ \frac{(\Gamma, x : (\mathsf{int}; \mathsf{reg}\ k)), P, F, S \vdash a : \tau; M, F', C}{(\Gamma, x : (\mathsf{int}; \mathsf{imm}\ n)), P, F, S \vdash a : \tau; M \cup \{k\}, F', C'}$$

$$\begin{aligned} \textit{where}\ \ & k \in \mathsf{Regs(int)} \setminus (\mathsf{Refer}\ \Gamma \cup P \cup F) \\ & C' = (R[k] := n; C) \end{aligned}$$

The resolution of an indirection $\mathsf{mem}\langle i, \mathsf{reg}\ n\rangle$ generates a memory load with index register $R[n]$ and offset $i$.

$$\textit{(conv-mem)}\ \frac{(\Gamma, x : (\sigma; A[\mathsf{reg}\ k])), P, F, S \vdash a : \tau; M, F', C}{(\Gamma, x : (\sigma; A[\mathsf{mem}\langle i, \mathsf{reg}\ n\rangle])), P, F, S \vdash a : \tau; M \cup \{k\}, F', C'}$$

$$\begin{aligned} \textit{where}\ \ & k \in \mathsf{Regs}(\sigma) \setminus (\mathsf{Refer}\ \Gamma \cup P \cup F) \\ & C' = (R[k] := M[R[n] + i]; C) \end{aligned}$$

There is also an operation that generates indirections by spilling a group of registers to memory. The register $k$ must be suitable to hold the standard representation of a tuple (a pointer to a contiguous area of memory) as indicated by $k \in \mathsf{RegisterNames}\times$. The *(spill)* rule is not applicable if there is no such register $k$. The rule chooses nondeterministically a set $X$ of registers to spill which does not interfere with the fixed registers $F$. If preserved registers are spilled the corresponding reloads are scheduled in the $S$ component.

$$\textit{(spill)}\ \frac{\tilde{\Gamma}, (P \setminus X) \cup \{k\}, F, \langle k, i_1 \dots i_p \rangle S \vdash a : \tau; M, F', C}{\Gamma, P, F, S \vdash a : \tau; (M \setminus X) \cup \{k\}, F', C'}$$

$$\begin{array}{ll} \textit{where} & \tilde{\Gamma} = \Gamma[\mathsf{reg}\ i_j := \mathsf{mem}\langle j, \mathsf{reg}\ k\rangle \mid 1 \le j \le n] \\ & X = \{i_1, \dots i_n\} \\ & X \cap F = \emptyset \\ & X \cap P = \{i_1, \dots, i_p\}, 0 \le p \le n \\ & k \in \mathsf{Regs}(\times) \setminus (\mathsf{Refer}\ \Gamma \cup P \cup F) \\ & C' = \ (R[k] := \mathsf{Allocate}(|X|); \\ & \qquad M[R[k]+0] := R[i_1]; \dots; M[R[k]+n-1] := R[i_n]; C) \end{array}$$

The notation $\Gamma[\mathsf{reg}\ i_j := \mathsf{mem}\langle j, \mathsf{reg}\ k\rangle \mid 1 \le j \le n]$ denotes the textual replacement of all occurrences of $\mathsf{reg}\ i_j$ in implementation types mentioned in $\Gamma$ by $\mathsf{mem}\langle j, \mathsf{reg}\ k\rangle$ for $1 \le j \le n$.

The corresponding inverse rule *(reload)* pops one reload entry from $S$.

$$\textit{(reload)}\ \frac{\Gamma, (P \setminus \{k\}) \cup \{i_1, \dots, i_p\}, F, S \vdash a : \tau; M, F', C}{\tilde{\Gamma}, P, F, \langle k, i_1 \dots i_p \rangle S \vdash a : \tau; M, F', C'}$$

$$\begin{array}{ll} \textit{where} & \tilde{\Gamma} = \Gamma[\mathsf{reg}\ i_j := \mathsf{mem}\langle j, \mathsf{reg}\ k\rangle \mid 1 \le j \le p] \\ & C' = (R[i_1] := M[R[k]+0]; \dots; R[i_p] := M[R[k]+p-1]; C) \end{array}$$

**Computation rules** The first computation rule deals with lambda abstraction. The type assumptions are divided in those for the free variables of the function $\Gamma$ and those for the continuation $\Delta$. The function's body $a_1$ is processed with $\tilde{\Gamma}$ where some free variables are relocated into the closure, a set $P'$ of preserved registers as determined by the call sites of the function, and a set of fixed variables $F'$ that contains those fixed registers that are referred to from the assumption $\Gamma$. Also, the register $m$ on the function arrow must match the register which is assumed to hold the closure while translating the body of the abstraction. It is not necessary that $m = k$, where $k$ is the register where the closure is allocated. Finally, the $\mathsf{let}$'s body $a_2$ is processed with $\Delta$.

$$\textit{(let-abs)}\ \frac{\begin{array}{c}(\tilde{\Gamma}, x_2 : \tau_2, c : (\mathsf{cont}\ F''\ ; l)), P', F', \varepsilon \vdash a_1 : \tau_1; M', F'', C_1 \\ (\Delta, x_1 : (\tau_2 \xrightarrow{F', P', M', F'', m} \mathsf{cont}\ l\ \tau_1; \mathsf{reg}\ k)), P, F, S \vdash a_2 : \tau_0; M, F', C_2\end{array}}{(\Gamma, \Delta), P, F, S \vdash \mathsf{let}\ x_1 = \lambda x_2.a_1\ \mathsf{in}\ a_2 : \tau_0; M \cup \{k\}, F', C'}$$

$$\begin{array}{ll} \textit{where} & F \cap \mathsf{Refer}\ \Gamma \subseteq F' \subseteq \mathsf{Refer}\ \Gamma \\ & k \in \mathsf{Regs}(\rightarrow) \setminus (\mathsf{Refer}\ (\Gamma, \Delta) \cup P \cup F) \\ & m \in \mathsf{Regs}(\rightarrow) \setminus (\mathsf{Refer}\ \Gamma \cup P' \cup F') \\ & \tilde{\Gamma} = \Gamma[\mathsf{reg}\ i_j := \mathsf{mem}\langle j, \mathsf{reg}\ m\rangle \mid 1 \le j \le n] \\ & \{i_j\} = \mathsf{Refer}\ \Gamma \setminus F', |\{i_j\}| = n \\ & C' = \ (\mathsf{Goto}\ t_2; t_1 : C_1; t_2 : R[k] = \mathsf{Allocate}(n+1); \\ & \qquad M[R[k]] := t_1; M[R[k]+1] := R[i_1]; \dots; M[R[k]+n] := R[i_n]; C_2) \end{array}$$

All registers that do not become fixed in the function must be evacuated into the closure for the function which is composed in $R[k]$. Since the continuation (which is located in $l$) can be handled like any other value, we invent a continuation identifier $c$ and bind it to the continuation. This is a drawback of A-normal form in comparison to continuation-passing style where continuation identifiers are explicit.

Next, we consider a typical primitive operation.

$$\textit{(let-add)}\ \frac{(\Gamma, x_1 : (\mathsf{int}; \mathsf{reg}\ k)), P, F, S \vdash a : \tau, M, F', C}{(\Gamma, x_2 : (\mathsf{int}; \mathsf{reg}\ i), x_3 : (\mathsf{int}; \mathsf{reg}\ j)), P, F, S \vdash a' : \tau, M \cup \{k\}, F', C'}$$

$$\begin{array}{ll} \textit{where} & k \in \mathsf{Regs(int)} \setminus (\mathsf{Refer}\ \Gamma \cup P \cup F) \\ & C' = (R[k] := R[i] + R[j]; C) \\ & a' = \mathsf{let}\ x_1 = x_2 + x_3\ \mathsf{in}\ a \end{array}$$

In addition, we could include a rule for constant propagation (in the case where the arguments are $\mathsf{imm}\ n_1$ and $\mathsf{imm}\ n_2$) and also rules to exploit immediate addressing modes if the processor provides for these.

Next, we consider the application of a function.

$$\textit{(let-app)}\ \frac{(\Gamma, x_1 : \tau_1), P, F'', S \vdash a : \tau, M, F', C}{\Gamma', P, F, S \vdash a' : \tau, M \cup M' \cup \{j, k\}, F', C'}$$

$$\begin{array}{ll} \textit{where} & P \cup \{i_1, \ldots, i_p\} \subseteq P', F' \subseteq F \\ & \{i_1, \ldots, i_n\} = \mathsf{Refer}\ \Gamma \setminus F', |\{i_j\}| = n \\ & \{j, k\} \subseteq \mathsf{Regs}(\rightarrow) \setminus (\mathsf{Refer}\ \Gamma, x_1 : \tau_1 \cup P \cup F), j \neq k \\ & \Gamma' = (\Gamma, x_2 : (\tau_2 \xrightarrow{F', P', M', F'', i} \mathsf{cont}\ (\mathsf{reg}\ j)\ \tau_1; \mathsf{reg}\ i), x_3 : \tau_2) \\ & a' = \mathsf{let}\ x_1 = x_2\ @\ x_3\ \mathsf{in}\ a \\ & C' = \ (R[j] := \mathsf{Allocate}(n - p + 1); M[R[j] + 0] := t; \\ & \qquad M[R[j] + 1] := R[i_{p+1}]; \ldots; M[R[j] + n - p] := R[i_n]; \\ & \qquad R[k] := M[R[i] + 0]; \mathsf{Goto}\ R[k]; \\ & \qquad t : R[i_{p+1}] := M[R[j] + 1]; \ldots; R[i_n] := M[R[j] + n - p]; C) \end{array}$$

The memory allocation in this rule saves values that are accessed by the continuation $a$. The preservation of the remaining registers is left to the callee by placing them $\{i_1, \ldots, i_p\}$ in the set of preserved registers $P'$. $R[j]$ points to the continuation closure. The sole purpose of the $\mathsf{cont}\ (\mathsf{reg}\ j)\ \tau_1$ construction lies in the transmission of the location of the continuation. The set of currently preserved registers must be a subset of the set of registers preserved by the function. Conversely, the set of currently fixed registers must contain the set of fixed registers demanded by the function. The continuation has to fix registers as indicated by the annotation $F''$ of the function type.

The $i$ on the function arrow indicates the register where the function body expects its closure. It must coincide with the register in which the closure actually is.

**Return rules** Finally, we need to consider rules that pass a value to the continuation. The most simple rule just returns the value of a variable. Due to the conversion rules, we can rely on $x : \tau$ already being placed in the location where the continuation expects it. All return rules expect that their reload list is empty.

$$\textit{(ret-var)}\ \frac{k \in \mathsf{Regs}(\rightarrow) \setminus (\mathsf{Refer}\ x : \tau, c : (\mathsf{cont}\ F\ ; \mathsf{reg}\ i) \cup P \cup F)}{(x : \tau, c : (\mathsf{cont}\ F\ ; \mathsf{reg}\ i)), P, F, \varepsilon \vdash x : \tau, \{k\}, F, C}$$

$$\textit{where}\ \ C = (R[k] := M[R[i] + 0]; \mathsf{Goto}\ R[k])$$

In this rule, the current continuation identifier $c$ indicates that register $i$ contains the continuation closure. As with any closure, its zero-th component contains the code address.

The final rule specifies a tail call to another function.

$$\textit{(ret-app)}\ \frac{P \subseteq P', F' \subseteq F, k \in \mathsf{Regs}(\rightarrow) \setminus (\{j, i\} \cup \mathsf{Refer}\ \tau_2 \cup P \cup F)}{\Gamma, P, F, \varepsilon \vdash x_1\ @\ x_2 : \tau_1, \{k\}, F, (R[k] := M[R[i] + 0]; \mathsf{Goto}\ R[k])}$$

$$\begin{aligned}\textit{where}\ \ \Gamma = {} & (x_1 : (\tau_2 \xrightarrow{F', P', M', F''} \mathsf{cont}\ (\mathsf{reg}\ j)\ \tau_1; \mathsf{reg}\ i), \\ & x_2 : \tau_2, c : (\mathsf{cont}\ F\ ; (\mathsf{reg}\ j)))\end{aligned}$$

There is neither a return term nor a return rule for addition, because the allocation properties of $\mathsf{let}\ x = y + z\ \mathsf{in}\ x$ are identical to those of $y + z$, if the latter was a legal return term.

## 4 Properties

In this section, we formalize some of the intuitive notions introduced in the preceding sections. First, we show that preserved registers really deserve their name.

**Theorem 1.** *Suppose $\Gamma, P, F, S \vdash a : \tau; M, F_1, C$ and the processor is in state $\langle C, \overline{R}, \overline{M} \rangle$. For each register $r \in P$:*

*Suppose $c : (\mathsf{cont}\ F''\ ; \mathsf{reg}\ w) \in \Gamma$, $y = R[r]$, $c = R[w]$, and $\langle C, \overline{R}, \overline{M} \rangle \overset{*}{\mapsto} \langle C', \overline{R}', \overline{M}' \rangle$. If $R'[w] = c$ and $C'$ is a suffix of $C$ such that $\Gamma', P', F', S' \vdash a' : \tau; M', F_1', C'$ and in the derivation steps between $a$ and $a'$ the reload component always has $S$ as a suffix then $R'[r] = y$.*

The reference to the continuation $c$ ensures that both machine states belong to the same procedure activation, by the uniqueness of addresses returned by $\mathsf{Allocate}(n)$. It provides the only link between the two machine states. If we dropped this requirement we would end up comparing machine states from different invocations of the same function and we could not prove anything. The condition on the reload component means that arbitrary spills are allowed between $p$ and $p'$, but reloads are restricted not to remove the reload record that was top-level at $p$. In other words, $S$ serves as a low-water mark. Our main interest will be in the case where $S = S' = \varepsilon$, $a$ is the body of a function, and $a'$ is a return term. In this case, the theorem says that registers mentioned in $P$ are preserved across function calls.

This theorem can be proved by induction on the number of control transfers in $\langle C, \overline{R}, \overline{M} \rangle \overset{*}{\mapsto} \langle C', \overline{R}', \overline{M}' \rangle$ and then by induction on the derivation.

Next, we want to formalize a property for $F$. A value stored in $f \in F$ will remain there unchanged as long as the variable binding that $f$ belongs to is in effect or reachable through closures or the continuation.

As a first step, we define a correspondence between an environment $\Gamma$ and a state of the CEK machine (cf. Sec. 2.2).

**Definition 2.** *$\Gamma \vdash (a, \rho, \kappa)$ if*

1. *there exist $P, F, S, M, F_1, C$ such that $\Gamma, P, F, S \vdash a : \tau; M, F_1, C$;*
2. *$x : \tau$ in $\Gamma$ implies $x \in \mathsf{dom}(\rho)$ and $\rho(x) \in \mathsf{TSem}\ \tau$;*
3. *if $c$ : $(\mathsf{cont}\ F''\ ; \mathsf{reg}\ w)$ in $\Gamma$ then there are $\rho$, $x$, and $a$ such that $\kappa = \langle \rho, x, a, \kappa' \rangle$. Otherwise $\kappa = \langle\rangle$.*

Unfortunately the connection between $c$ and $\kappa$ is not very deep. We cannot properly relate $c$ to $\kappa$ since the "type" of $c$ does not refer to an environment. In fact, $c$ and its type cannot refer to a specific environment $\Gamma'$ because $a$ may be called from several places with different environments. Therefore, the type of the return environment of the continuation must be polymorphic. The function $\mathsf{TSem}\ \tau$ maps an implementation type to a subset of $\mathsf{Val}$.

$$
\begin{aligned}
&\mathsf{TSem}\ (\mathsf{int}; l) = \{\mathsf{Num}\ (n) \mid n \text{ is an integer}\} \\
&\mathsf{TSem}\ (\tau_2 \xrightarrow{F,P,M,F'} \mathsf{cont}\ l'\ \tau_1; l) = \\
&\qquad \{\mathsf{Fun}\ (\rho', \lambda y.a) \mid \\
&\qquad \forall z \in \mathsf{TSem}\ \tau_2. \\
&\qquad (a, \rho'[y \mapsto z], \langle\rangle) \overset{*}{\mapsto} (x, \rho'', \langle\rangle) \text{ such that } \rho''(x) \in \mathsf{TSem}\ \tau_1 \text{ or} \\
&\qquad (a, \rho'[y \mapsto z], \langle\rangle) \overset{*}{\mapsto} (x + w, \rho'', \langle\rangle) \text{ and } \tau_1 = (\mathsf{int}; l')\}
\end{aligned}
$$

However, to formalize reachability through closures and continuations and link this concept with the environment, we need a stronger notion than $\Gamma \vdash (a, \rho, \kappa)$.

What we can actually prove by inspection of the rules is a much weaker theorem.

**Theorem 2.** *Suppose $\Gamma, P, F, S \vdash a : \tau; M, F_1, C$ and the processor is in state $\langle C, \overline{R}, \overline{M} \rangle$. For each $r \in F$:*

*Suppose $y = R[r]$ and $\langle C, \overline{R}, \overline{M} \rangle \overset{*}{\mapsto} \langle C', \overline{R}', \overline{M}' \rangle$ such that $\Gamma', P', F', S' \vdash a' : \tau; M', F_1', C'$ and there is no intermediate state with a corresponding derivation step. If furthermore $r \in F'$ then $R'[r] = y$.*

Finally, we establish a formal correspondence between steps of the CEK machine and steps of the translated machine program. To this end, we need a notion of compatibility between a CEK state and a machine state, $\Gamma \vdash (a, \rho, \kappa) \cong (C, \overline{R}, \overline{M})$.

**Definition 3.** *Suppose* $\Gamma, P, F, S \vdash a : \tau; M, F_1, C$.

$\Gamma \vdash (a, \rho, \kappa) \cong (C, \overline{R}, \overline{M})$ *if for all* $x : \tau$ *in* $\Gamma$:

- *if* $\tau = (\mathsf{int}; \mathsf{imm}\ n)$ *then* $\rho(x) = \mathsf{Num}\ (n)$ *(the value is not represented in the machine state);*
- *if* $\tau = (\mathsf{int}; \mathsf{reg}\ r)$ *then there exists an integer* $n$ *s.t.* $\rho(x) = \mathsf{Num}\ (n)$ *and* $R[r] = n$*;*
- *if* $\tau = (\mathsf{int}; \mathsf{mem}\langle j, \mathsf{reg}\ r\rangle)$ *then there exists* $n$ *s.t.* $\rho(x) = \mathsf{Num}\ (n)$ *and* $M[R[r] + j] = n$*;*
- *if* $\tau = (\mathsf{int}; \mathsf{mem}\langle j_k, \dots \mathsf{mem}\langle j_0, \mathsf{reg}\ r\rangle\rangle)$ *then there exist* $n, i_0, \dots, i_k$ *s.t.* $\rho(x) = \mathsf{Num}\ (n)$ *and* $i_k = n$, $i_\nu = M[i_{\nu-1} + j_\nu]$ *for* $1 \leq \nu \leq k$, *and* $i_0 = R[r]$*;*
- *if* $\tau = (\tau_2 \xrightarrow{F', P', M', F_1'} \mathsf{cont}\ (\mathsf{reg}\ s)\ \tau_1; \mathsf{reg}\ r)$ *then there exists* $\rho', y, a'$ *s.t.* $\rho(x) = \mathsf{Fun}\ (\rho', \lambda y.a')$ *and for all* $z \in \mathsf{TSem}\ \tau_2$ $(a', \rho'[y \mapsto z], \langle\rangle) \overset{*}{\mapsto} (x', \rho'', \langle\rangle)$ *where* $\rho''(x') \in \mathsf{TSem}\ \tau_1$, $M[R[r]]$ *holds the address of* $C'$ *such that*

$$(\Gamma', y : \tau_2, c : (\mathsf{cont}\ F_2\ ; \mathsf{reg}\ s)), P', F', S' \vdash a' : \tau_1; M', F_1', C'$$

*which starts with*

$$\Gamma'', P'', F'', S'' \vdash x' : \tau_1; M', F_1', C''$$

*and for each CEK state* $(C', \overline{R}', \overline{M}') \overset{*}{\mapsto} (C'', \overline{R}'', \overline{M}'')$ *where* $R'[s] = R''[s]$ *we have that* $(\Gamma', y : \tau_2, c : (\mathsf{cont}\ F_2\ ; l)) \vdash (a', \rho'[y \mapsto z], \kappa') \cong (C', \overline{R}', \overline{M}')$ *and* $\Gamma'' \vdash (x', \rho'', \kappa') \cong (C'', \overline{R}'', \overline{M}'')$*;*
- *if* $\tau = (\mathsf{cont}\ F_2\ ; \mathsf{reg}\ r)$ *then* $\kappa = \langle \rho', x', a', \kappa' \rangle$ *such that* $\Gamma', P', F', S' \vdash a' : \tau'; M', F_1', C'$ *and* $M[R[r]]$ *holds the address of* $C'$.

**Theorem 3.** *Suppose* $\Gamma, P, F, S \vdash a : \tau; M, F_1, C$.

*If* $\Gamma \vdash (a, \rho, \kappa) \cong (C, \overline{R}, \overline{M})$ *and* $(a, \rho, \kappa) \overset{*}{\mapsto} (a', \rho', \kappa')$ *where* $\kappa$ *is a suffix of* $\kappa'$ *then* $(C, \overline{R}, \overline{M}) \overset{*}{\mapsto} (C', \overline{R}', \overline{M}')$ *and there exist* $\Gamma'$, $P'$, $F'$, $S'$, $\tau'$, $M'$, *and* $F_1'$ *such that* $\Gamma', P', F', S' \vdash a' : \tau', M', F_1', C'$ *and* $\Gamma' \vdash (a', \rho', \kappa') \cong (C', \overline{R}', \overline{M}')$.

## 5 Related Work

Compiling with continuations has already a long history. Steele's Rabbit compiler [21] has pioneered compilation by first transforming source programs to continuation-passing style and then transforming it until the assembly code can be read of directly. Also the Orbit compiler [13] and other successful systems [2, 3, 12] follow this strategy.

Recently, there has been some interest in approaches which do not quite transform the programs to continuation-passing style. The resulting intermediate language has been called nqCPS[1], A-normal form [5], monadic normal form [9], etc. These languages are still direct style languages, but have the following special features

[1] This term has been coined by Peter Lee [14] but it has never appeared in a published paper.

1. the evaluation order (and hence the control flow) is made explicit;
2. all intermediate results are named;
3. the structure of an expression makes the places obvious where serious computations are performed (i.e., where a continuation is required in the implementation).

Another related line of work is boxing analysis (e.g., [10, 18, 19, 22]). Here the idea is to try to avoid using the inefficient boxed representation of values in polymorphic languages. We believe that our system is powerful enough so that (an polymorphic extension of) it can also express the necessary properties.

Representation analysis [4] which is used in conjunction with region inference has one phase (their section 5 "Unboxed Values") whose concerns overlap with the goals of our system. Otherwise, their system is concerned with finding the number of times a value is put into a region, the storage mode of a value which determines whether a previous version may be overwritten, or the physical size of a region.

Typed assembly language [15] is an approach to propagate polymorphic type information throughout all phases of compilation. This work defines a fully typed translation from the source language (a subset of core ML) down to assembly language in four stages: CPS conversion, closure conversion, allocation, and code generation. As it is documented, the allocation phase takes the conventional fully boxed approach to allocating closures and tuples. It remains to investigate whether the allocation phase operates on the right level of abstraction to take the decisions that we are interested in controlling with our approach.

## 6 Conclusion

We have investigated an approach to specify decisions taken in the code generation phase of a compiler using a non-standard type system. The system builds on simple typing and uses a restricted version of A-normal form as its source language. We have defined a translation that maps the source language into abstract assembly language and have verified some properties of the translation. One goal of the approach is the verification and specification of code generators by adding constraints to our system that make the typing judgements deterministic.

In the course of the work on this system, we have learned a number of lessons on intermediate languages for compilers that want to apply advanced optimization techniques (like unboxing, lightweight closure conversion, and so on):

It is essential to start from an intermediate language that clearly distinguishes serious computations (that need a continuation) from trivial ones (that yield values directly). Otherwise, rules like the *(spill)* rule could not be applied immediately before generating the machine code for the actual function call.

It is also essential that A-normal form sequentializes the computation. For unrestricted direct-style expressions the control flow is sufficiently different from the propagation of information in typing judgements to make such a formulation awkward, at best. For example, it would be necessary to thread the typing assumptions through the derivation according to the evaluation order.

Also, in an unrestricted direct-style term it is sometimes necessary to perform a resource operation (for example, spilling or representation conversion) on return from the evaluation of an expression. This leads to a duplication of typing rules, one set determining the operations on control transfer into the expression and another set for the operations on return from the expression. In A-normal form, returning from one expression is always entering the next expression, so there is no restriction in having only the first set of these rules.

On the negative side, we found that A-normal form does not supply sufficient information when it comes to naming continuations. In contrast to continuation-passing style, our translation has to invent continuation variables in order to make the continuation visible to the resource allocation machinery. It would be interesting to investigate a similar system of implementation types for an intermediate language in continuation-passing style and to establish a formal connection between the two.

Another point in favor of continuation-passing style is the allocation of continuation closures. Presently this allocation clutters the rule for function application *(let-app)*. A system based on continuation-passing style might be able to decompose the different tasks present in *(let-app)* into several rules.

Finally, in places the rules are rather unwieldy so that it is debatable whether the intermediate language that we have chosen is the right level of abstraction to take the decisions that we are interested in. Imposing the system, for example, on the allocation phase of a system like that of Morrisett et al. [15] might in the end lead to a simpler system and one where the interesting properties can be proved more easily.

Beyond the work reported in this paper, we have already extended the framework to conditionals, and to sum and product types. The current formulation does not allow for unboxed function closures. This drawback can be addressed at the price of a plethora of additional rules. Future work will address the incorporation of polymorphic types and investigate the possibilities of integrating our work with typed assembly language.

**Acknowledgment** Many thanks to the reviewers. Their detailed comments helped to clean up the presentation substantially.

## References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
3. Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In POPL1989 [16], pages 293–302.
4. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, Fla., January 1996. ACM Press.

5. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.
6. Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.
7. David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on Lisp and Functional Programming*, pages 28–38, 1986.
8. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
9. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In POPL1994 [17], pages 458–471.
10. Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In POPL1994 [17], pages 213–226.
11. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 1991. ACM Press.
12. Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In POPL1989 [16], pages 281–292.
13. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, July 1986. Proc. Sigplan '86 Symp. on Compiler Construction.
14. Peter Lee. The origin of nqCPS. Email message, March 1998.
15. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, USA, January 1998. ACM Press.
16. *16th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989. ACM Press.
17. *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OG, January 1994. ACM Press.
18. Zhong Shao. Flexible representation analysis. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, pages 85–98, Amsterdam, The Netherlands, June 1997. ACM Press, New York.
19. Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, USA, June 1995. ACM Press.
20. Paul Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, January 1997.
21. Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.
22. Peter Thiemann. Polymorphic typing and unboxed values revisited. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 24–35, La Jolla, CA, June 1995. ACM Press, New York.
23. Philip Wadler. Is there a use for linear logic? In Paul Hudak and Neil D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 255–273, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).