ROPAS
Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
1999-2
November 15, 1999

Reconstructing the Types of Stack-Machine Codes

Oukseh Lee and Kwangkeun Yi
{cookcu; kwang}@ropas.kaist.ac.kr

Abstract

It is frequently needed to compile stack-machine codes into register-machine codes. One important optimization in such compilers is reducing the stack access overhead. But an effective mapping of stack values into registers is not straightforward. In this article, we present a formal yet effective technique of inferring the two types of each stack value. We infer the type of a stack value when it is pushed (*push-type*) and the type when it is used (*pop-type*). These two type information is safely estimated across the basic blocks by a global data-flow analysis. Using this type information, we can safely use as many typed registers as possible in storing stack values. We implemented our analysis for a real compiler and its experiments show that the speed-up is 5% to 24%.

1 Problem and Our Approach

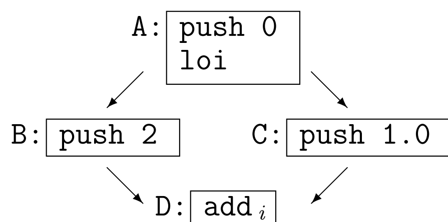
It is frequently needed to compile stack-machine codes into register-machine codes [ATCL+98, Ert96, TvSKS83]. Stack-machine codes are used widely as intermediate languages, while most computers are register machines.

One important optimization in such compilers is reducing the stack access overhead. Stack-machine codes usually access the stack for keeping temporary values. If we naively implement the stack in the target machine's main memory, the memory traffic (load and store) easily becomes the performance bottleneck.

But an effective mapping of stack values into registers is not straightforward. The problem is that the registers are typed, while the stacks are un-typed. Therefore, in order to utilize a typed register for a stack value, the value's type have to be determined. This problem is complicated when a stack value's type when it is pushed is different from the type when it is used. When these two types are the same, we can use a typed register to store the value.

In this article, we present a formal yet effective technique of inferring the two types of each stack value. We infer the type of a stack value when it is pushed (*push-type*) and the type when it is used (*pop-type*). These two type information is safely estimated across the basic blocks by a global data-flow analysis. Using this type information, we can safely use as many typed registers as possible in storing stack values.

As an example of our type analysis and its use for mapping stack values to typed registers, consider the control flows of four code segments `A`, `B`, `C`, and `D`:



[0]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

The possible control-flows are `A`→`B`→`D` and `A`→`C`→`D`. The first value popped by `D` is pushed as an integer (`2`) or a real number (`1.0`), and the value is popped as an integer because the $\texttt{add}_i$ command is for *i*nteger-addition. Thus the common storage that can contain both an integer and a real number is the stack. The second value popped by `D` is considered an integer but was pushed as a typeless word at `A`. (because `A` pushes the value from the memory of address `0` (`M[0]`) by the load-indirect command `loi`.) Thus the common storage is an integer register. By these results, we store `M[0]` to an integer register, and `2` and `1.0` to the stack, and `D` pops one value from the integer register and the other from the stack.

We implemented our analysis for a real compiler and its experiments show that the speed-up is 5% to 24%.

No other works have applied such an extensive data-flow analysis for inferring the types of stack values. Existing works either exploit the types of stack values only within the basic blocks [vS84], or use the types across the basic blocks but assume that no ill-typed uses occur [Ert95, Ert96, Kna93, MCGW98].

2 The Source Language

The source language (stack-machine language) is shown in Figure 1. The language is simplified for presentation brevity. In reality of our implementation, the source language is the EM [TvSKS83]. The values of the language are one-word integers or one-word real numbers. Boolean values are treated as integers. The stack is the sequence of typeless words. Integers and real numbers are used as encoded typeless words in stack operations. Thus ill-typed programs are possible to work well.

A state-transition semantics of the source language is also shown in Figure 1, which describes an one-step evaluation. The semantics of a program $\wp$ is the sequence of the $\rightarrow$ relations from the initial state. A state (c, H, σ) consists of the command c pointed to by the current program counter, a heap H and a stack σ. The initial state is $(0{:}\, e \in \wp, \emptyset, \epsilon)$. Command labels are distinct and consecutive positive integers from 0 and plays a role as a program point. Each stack value has its push-point that indicates which command in the program pushed it. We attach an integer index to a push-point because some commands (e.g. $\texttt{dup}_\tau$) push more than one value. We shortly write l_i for a push-point $\langle l, i\rangle$.

3 Push-Type Analysis

We first approximate the push-types of stack values by using the abstract interpretation framework [CC77, CC92]. In this framework, we define an abstract semantics of stack-machine codes, by which we can finitely approximate at compile-time the types of stack values when they are pushed at run-time.

The correctness of such an abstract semantics is achieved by establishing a sound relation between the abstract semantics and a collecting semantics which collects all the possible stack values that will occur at run-time after each command's execution.

The collecting semantics is shown in Figure 2. It simply collects all the possible heaps and stacks for each program point [CC77]. Note that the collecting semantics are not computable at compile-time. For example, codes with an infinite loop can have infinitely many stack configurations.

We therefore need to define a finite approximation of the collecting semantics, so that the execution of the source program based on the finite semantics always terminates and becomes our compile-time analysis.

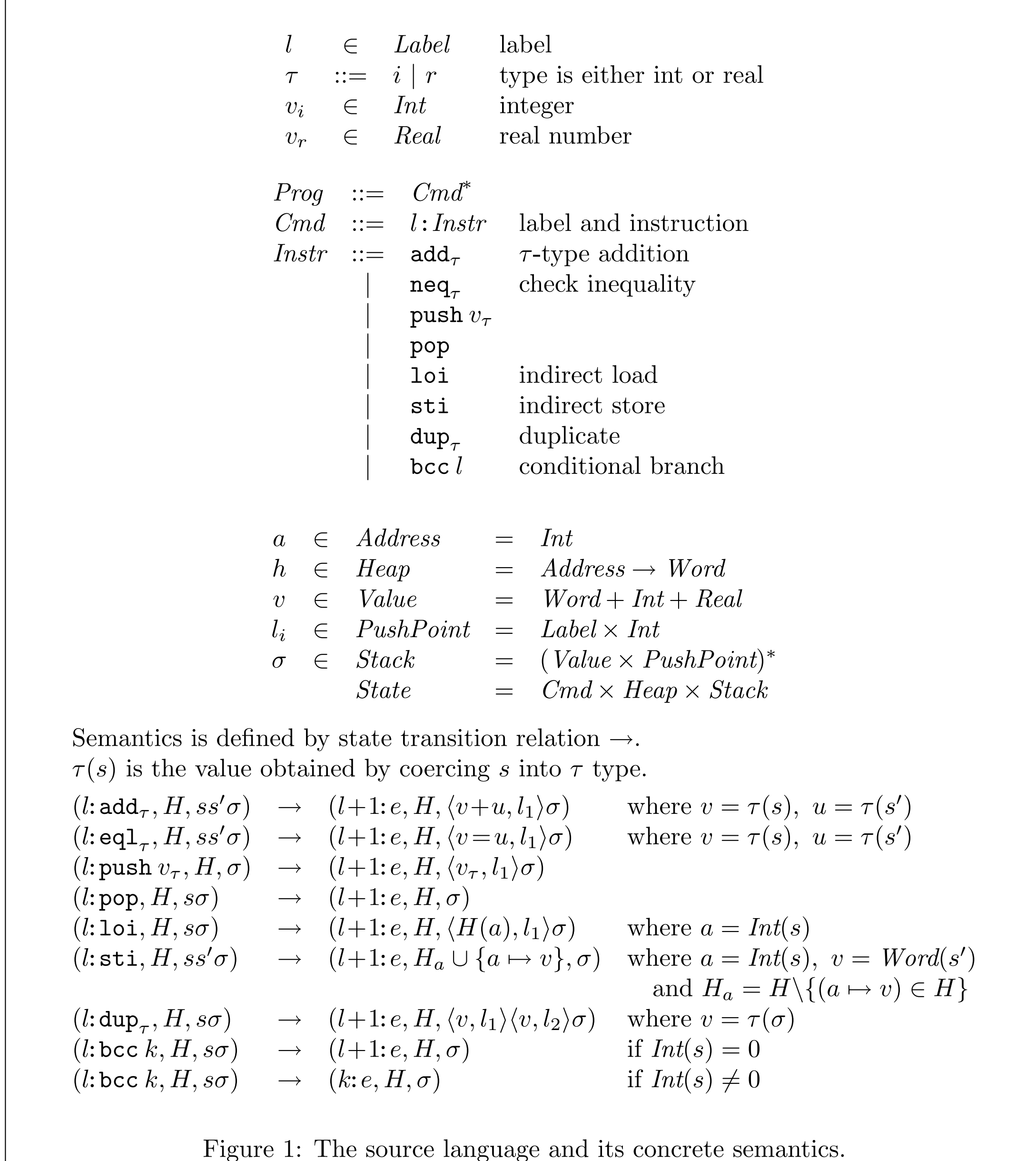


Figure 1: The source language and its concrete semantics.

We finitely approximate a set of heaps and stacks by a single stack-type. The set $\widehat{\mathcal{D}}$ of stack-types is a lattice whose elements are the objects we compute during our analysis.

Definition 1 ($\widehat{\mathcal{D}}, \sqsubseteq$). *The stack-type $\widehat{\sigma}$ is defined as*

$$
\begin{array}{lcll}
t & \in & Type & ::= \ \mathbf{u} \mid \mathbf{i} \mid \mathbf{n} \mid \mathbf{r} \\
\widehat{\sigma} & \in & \widehat{\mathcal{D}} & ::= \ \bot \mid \langle t, L\rangle^* \top
\end{array}
$$

where $L \in \wp(PushPoint)$, and the stack-type has the following order:

$$
\begin{array}{l}
\bot \sqsubseteq \widehat{\sigma} \sqsubseteq \top \ \textit{for all}\ \widehat{\sigma} \in \widehat{\mathcal{D}} \\
\langle t, L\rangle \widehat{\sigma} \sqsubseteq \langle t', L'\rangle \widehat{\sigma}' \ \textit{if}\ t \sqsubseteq_t t',\ L \subseteq L' \ \textit{and}\ \widehat{\sigma} \sqsubseteq \widehat{\sigma}'
\end{array}
$$

The collecting semantics is the least fix-point of function $\mathcal{G}$:

$$\mathcal{G} : (Cmd \to \mathcal{D}) \to (Cmd \to \mathcal{D}) \text{ where } \mathcal{D} = \wp(Heap \times Stack)$$

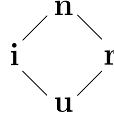
$$\mathcal{G}\ \mathcal{E}\ c = \mathcal{F}\ c\ \left(\cup_{c' \in pre(c)} \mathcal{E}(c')\right)$$

where

$$\mathcal{F}\, c\, X = \{(H', \sigma') \mid (c, H, \sigma) \to (c', H', \sigma'),\ (H, \sigma) \in X\}$$
$$pre(l{:}\, e) = \{l-1{:}\, e' \in \mathcal{P}\} \cup \{k{:}\, \texttt{bcc}\ l \in \mathcal{P}\}$$

Figure 2: Collecting semantics

where the orders of types are $\mathbf{u} \sqsubseteq_t \mathbf{i} \sqsubseteq_t \mathbf{n}$ *and* $\mathbf{u} \sqsubseteq_t \mathbf{r} \sqsubseteq_t \mathbf{n}$ *which is represented by:*



Our finite approximation of a set of heaps and stacks is done by the following steps. We completely approximate the heap as unknown. We approximate each stack value by its type. Then a stack becomes a sequence of value types. A set of type sequences is then approximated into a single type sequence by separately joining the types and the push-points in the same-depth cells of the stacks. The join of two types is defined by the partial order between types. The orders of types ($\sqsubseteq_t$) is determined by the storable values of the storage of a type. An integer register ($\mathbf{i}$) can store integers or typeless words. A real-type register ($\mathbf{r}$) can store real numbers or typeless words. The stack ($\mathbf{n}$) can store all values. A memory cell in the heap ($\mathbf{u}$) stores only typeless words. The set-inclusion order of these storable values is the order of types. In formal terms, the finite approximation is defined by the abstraction function α from a set of heaps and stacks to a stack type. The abstraction function α establishes a Galois connection paired with the concretization function γ:

Definition 2 $(\mathcal{D}, \subseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (\widehat{\mathcal{D}}, \sqsubseteq)$ *is a Galois connection for*

$$\alpha : \mathcal{D} \to \widehat{\mathcal{D}} \quad \text{such that } \alpha(X) = \bigsqcup_{(H,\sigma)\in X} \vec{type}(\sigma)\top$$
$$\gamma : \widehat{\mathcal{D}} \to \mathcal{D} \quad \text{such that } \gamma(\widehat{\sigma}) = \left\{(H, \sigma) \mid \vec{type}(\sigma)\top \sqsubseteq \widehat{\sigma}, H \in Heap\right\}$$

where $\vec{type}(\langle v_1, l_1\rangle \cdots \langle v_n, l_n\rangle) = \langle type(v_1), \{l_1\}\rangle \cdots \langle type(v_n), \{l_n\}\rangle$ *and* $type(v) = \mathbf{i}, \mathbf{r}, \mathbf{u}$ *if* $v \in$ *Int, Real, Word, respectively.*

We define an abstract semantics $\widehat{\mathcal{G}}$ of commands over the abstract domain $\widehat{\mathcal{D}}$, such that the abstract single-step transiton operation $\widehat{\mathcal{F}}\, c$ for a command c is equal to $\alpha \circ (\mathcal{F}\ c) \circ \gamma$. See Figure 3.

Definition 3 The push-type analysis $\widehat{\text{PUSH}}_\wp$ *for the source program* $\wp$ *is* $\bigsqcup_{i\geq 0} \widehat{\mathcal{G}}^i(\bot^\wp)$ *where* $\bot^\wp$ *is the finite table from the program points of* $\wp$ *to* $\bot$.

Theorem 1 *The push-type analysis is a safe approximation of the collecting semantics, and the analysis always terminates for all finite input programs.*

The abstract semantics is the least fix-point of function $\widehat{\mathcal{G}}$:

$$\widehat{\mathcal{G}} : (Cmd \to \widehat{\mathcal{D}}) \to (Cmd \to \widehat{\mathcal{D}})$$

$$\widehat{\mathcal{G}}\ \widehat{\mathcal{E}}\ c = \widehat{\mathcal{F}}\ c\ \left(\sqcup_{c' \in pre(c)} \widehat{\mathcal{E}}(c')\right)$$

where

$$\begin{array}{ll}
\widehat{\mathcal{F}}\ (l\!:\!\mathtt{add}_\tau)\ \widehat{\sigma} = \langle \tau, \{l_1\}\rangle \widehat{\sigma}_{[2\sim]} & \widehat{\mathcal{F}}\ (l\!:\!\mathtt{push}\ v_\tau)\ \widehat{\sigma} = \langle \tau, \{l_1\}\rangle \widehat{\sigma} \\
\widehat{\mathcal{F}}\ (l\!:\!\mathtt{neq}_\tau)\ \widehat{\sigma} = \langle \mathbf{b}, \{l_1\}\rangle \widehat{\sigma}_{[2\sim]} & \widehat{\mathcal{F}}\ (l\!:\!\mathtt{pop})\ \widehat{\sigma} = \widehat{\sigma}_{[1\sim]} \\
\widehat{\mathcal{F}}\ (l\!:\!\mathtt{loi})\ \widehat{\sigma} = \langle \mathbf{u}, \{l_1\}\rangle \widehat{\sigma}_{[1\sim]} & \widehat{\mathcal{F}}\ (l\!:\!\mathtt{dup}_\tau)\ \widehat{\sigma} = \langle \tau, \{l_1\}\rangle\langle \tau, \{l_2\}\rangle \widehat{\sigma}_{[1\sim]} \\
\widehat{\mathcal{F}}\ (l\!:\!\mathtt{sti})\ \widehat{\sigma} = \widehat{\sigma}_{[2\sim]} & \widehat{\mathcal{F}}\ (l\!:\!\mathtt{bcc}\ l)\ \widehat{\sigma} = \widehat{\sigma}_{[1\sim]}
\end{array}$$

The tail operator $\widehat{\sigma}_{[d\sim]}$ (which removes top d types from $\widehat{\sigma}$) is:
$\bot_{[d\sim]} = \bot$, $\widehat{\sigma}_{[0\sim]} = \widehat{\sigma}$, $\top_{[d\sim]} = \top$, and $(\widehat{s}\widehat{\sigma})_{[d\sim]} = \widehat{\sigma}_{[d-1\sim]}$ if $d > 0$

Figure 3: Abstract semantics

PROOF. The analysis is sound, because collecting domain $\mathcal{D}$ and abstract domain $\widehat{\mathcal{D}}$ establish a Galois connection by α and γ, and $\widehat{\mathcal{F}}\ c$ is defined as $\alpha \circ (\mathcal{F}\ c) \circ \gamma$ [CC77]. The termination is guaranteed because $\widehat{\mathcal{G}}$ is monotone and for a given program $\wp$, the chain $\bot^\wp$, $\widehat{\mathcal{G}}(\bot^\wp)$, $\widehat{\mathcal{G}}^2(\bot^\wp)$, $\cdots$ is always finite [CC77]. □

4 Pop-Type Analysis

In order to find a single type that subsumes both the types as which a stack value is pushed and popped, we need to propagate the pop-types of commands up to their push-points.

The propagation rules and axioms are shown in Figure 4. In our rules, "$l_i \Downarrow t$" means that a value can be pushed as t-type at l_i, "$k_j \Uparrow s$" means that a vaule can be popped as s-type at k_j, and "$l_i \to k_j$" means that the value pushed at l_i can be popped at k_j. Consider, for example, the (*Anal*) rule. If a value can be pushed as t-type at l_i ($l_i \Downarrow t$) and the same value can be popped as s-type at k_j ($k_j \Uparrow s$ and $l_i \to k_j$), then the join $t \sqcup s$ of the two types becomes the type of both push-point l_i and pop-point k_j.

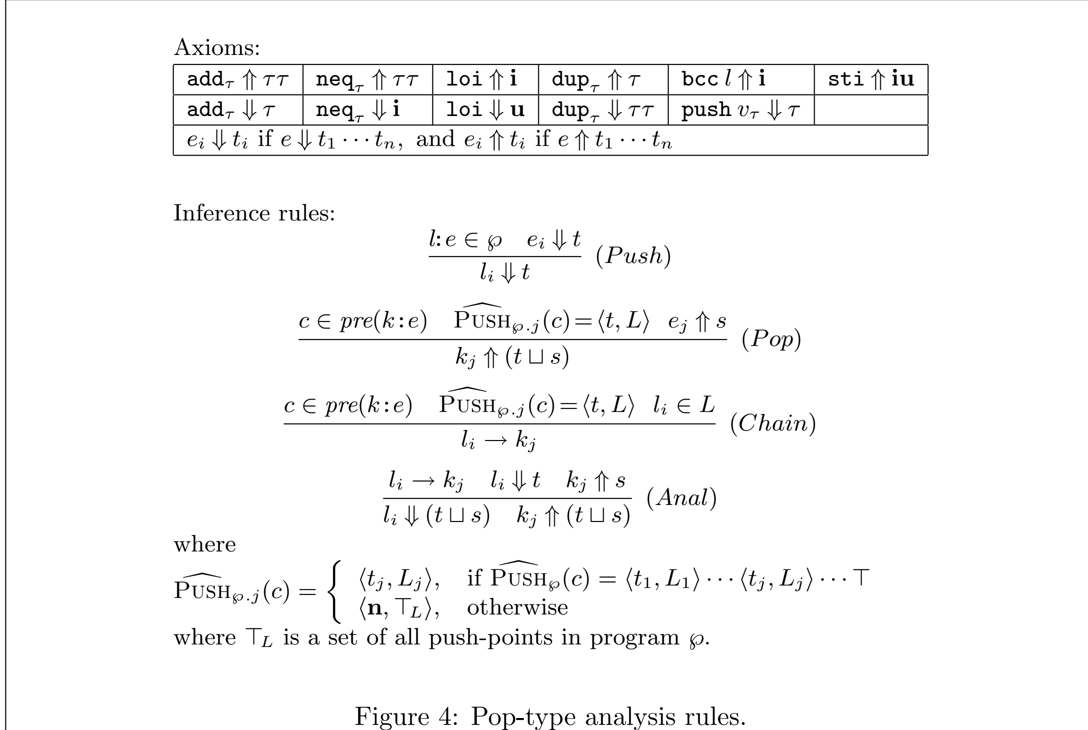
Definition 4 The pop-type analysis $\widehat{\mathrm{POP}}_\wp$ *for a program* $\wp$ *is obtaining* $l_i \to k_j$, $l_i \Downarrow t$ *and* $k_j \Uparrow s$ *for all push-point* l_i *and pop-point* k_j *of the program* $\wp$*, by repeatedly applying the inference rules of Figure 4 until no more* $l_i \to k_j$, $l_i \Downarrow t$ *and* $k_j \Uparrow s$ *can be added.*

Theorem 2 *The pop-type analysis is sound and always terminates.*

PROOF. The axioms asserts obviously safe push-type and pop-type of every command. The (*Push*), (*Pop*), and (*Chain*) rules generate safe relations that subsume the axioms and/or our safe push-type analysis results. The (*Anal*) rule always adds a new push-type (or pop-type) which subsumes prevously inferenced types. The analysis always terminates because the number of possible $l_i \to k_j$, $l_i \Downarrow t$ and $k_j \Uparrow t$ is finite for a given program. □

5 Implementation and Conclusion

We implemented the two analyses in a real compiler and found that it improves the generated code speed by 5–24%. The compilers are EM-to-PowerPC compilers developed by us as versions


Axioms:

$\mathtt{add}_\tau \Uparrow \tau\tau$	$\mathtt{neq}_\tau \Uparrow \tau\tau$	$\mathtt{loi} \Uparrow \mathbf{i}$	$\mathtt{dup}_\tau \Uparrow \tau$	$\mathtt{bcc}\, l \Uparrow \mathbf{i}$	$\mathtt{sti} \Uparrow \mathbf{iu}$
$\mathtt{add}_\tau \Downarrow \tau$	$\mathtt{neq}_\tau \Downarrow \mathbf{i}$	$\mathtt{loi} \Downarrow \mathbf{u}$	$\mathtt{dup}_\tau \Downarrow \tau\tau$	$\mathtt{push}\, v_\tau \Downarrow \tau$	
$e_i \Downarrow t_i$ if $e \Downarrow t_1 \cdots t_n$, and $e_i \Uparrow t_i$ if $e \Uparrow t_1 \cdots t_n$					

Inference rules:

$$\frac{l{:}e \in \wp \quad e_i \Downarrow t}{l_i \Downarrow t}\ (Push)$$

$$\frac{c \in pre(k{:}e) \quad \widehat{\text{PUSH}}_{\wp.j}(c) = \langle t, L\rangle \quad e_j \Uparrow s}{k_j \Uparrow (t \sqcup s)}\ (Pop)$$

$$\frac{c \in pre(k{:}e) \quad \widehat{\text{PUSH}}_{\wp.j}(c) = \langle t, L\rangle \quad l_i \in L}{l_i \to k_j}\ (Chain)$$

$$\frac{l_i \to k_j \quad l_i \Downarrow t \quad k_j \Uparrow s}{l_i \Downarrow (t \sqcup s) \quad k_j \Uparrow (t \sqcup s)}\ (Anal)$$

where

$$\widehat{\text{PUSH}}_{\wp.j}(c) = \begin{cases} \langle t_j, L_j\rangle, & \text{if } \widehat{\text{PUSH}}_{\wp}(c) = \langle t_1, L_1\rangle \cdots \langle t_j, L_j\rangle \cdots \top \\ \langle \mathbf{n}, \top_L\rangle, & \text{otherwise} \end{cases}$$

where $\top_L$ is a set of all push-points in program $\wp$.

Figure 4: Pop-type analysis rules.

with and without the type reconstruction. See Figure 5.

We present a practical analysis problem and design a sound analysis. We found type information of the stack can improve the quality of target codes. We designed an analysis that solved the problem, proved its safety, implemented it inside a real compiler, and showed that the analysis improved the generated code's speed up to 24%.

References

[ATCL+98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Canada, 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

[Ert95] Martin Anton Ertl. Stack caching for interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

	EM code size (KB)	analyses time (sec)[b]	execution time of codes (sec)[a]		speed improvement (%)
			generated without the type analyses	generated by the type analyses	
merge sort	3.2	0.08	241.56	210.38	12.1
129.compress[c]	139.4	0.71	3848.16	3638.74	5.4
132.ijpeg[c]	610.6	29.43	4896.10	3732.58	23.8
099.go[c]	1341.3	53.30	6281.24	5647.31	10.0

[a]PowerPC 604e 166 MHz, AIX 4.2.
[b]Sun Ultra Sparc 296 MHz, Solaris 2.6.
[c]SPEC95 benchmark programs, compiled by C-to-EM compiler (revision 3.10) of the Amsterdam Compiler Kit [TvSKS83].

Figure 5: Experimental result.

[Ert96] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universitaet Wien, Austria, 1996.

[Kna93] Peter J. Knaggs. *Practical and Theoretical Aspects of Fourth Software Development*. PhD thesis, School of Computing and Mathematics, University of Teesside, Middlesbrough, Cleveland, UK, March 1993.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *International Workshop on Types in Compilation*, 1998.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, Ed G. Kaizer, and Johan W. Stevenson. Description of a machine architecture for use with block structured languages. Technical report, Vrije Universiteit, Amsterdam, 1983.

[vS84] Hans van Staveren. The table driven code generator from the amsterdam compiler kit. Technical report, Vrije Universiteit, Amsterdam, The Netherlands, 1984.