Register Allocation by Proof Transformation*

Atsushi Ohori

School of Information Science
Japan Advanced Institute of Science and Technology
Tatsunokuchi, Ishikawa 923-1292, JAPAN; ohori@jaist.ac.jp

Abstract. This paper presents a proof-theoretical framework that accounts for the entire process of register allocation – liveness analysis is proof reconstruction (similar to type inference), and register allocation is proof transformation from a proof system with unrestricted variable accesses to a proof system with restricted variable access. In our framework, the set of registers acts as a "working set" of the live variables at each instruction step, which changes during the execution of the code. This eliminates the ad-hoc notion of "spilling". The necessary memory-register moves are systematically incorporated in our proof transformation process. Its correctness is a direct corollary of our construction; the resulting proof is equivalent to the proof of the original code modulo treatment of structural rules. The framework yields a clean and powerful register allocation algorithm. The algorithm has been implemented, demonstrating the feasibility of the framework.

1 Introduction

Register allocation is a process to convert an intermediate language to another language closer to machine code. Such a process should ideally be presented as a language transformation system that preserves the meaning of a program – both its static and dynamic semantics. These results will not only yield robust and systematic compiler implementation but also serve as a basis for reasoning about formal properties of register allocation process such as preservation of type safety, which will complement recent results on verifying type safety of low-level code, e.g. [11,6,7,8]. Unfortunately, however, it appear to be difficult to establish such results for existing methods of register allocation.

The most widely used method for register allocation is *graph coloring* [3,2]. It first performs liveness analysis of a given code and constructs an interference graph. It then solves the problem by "spilling" some nodes from the graph and finding a "coloring" of the remaining subgraph. Although it is effective and practically feasible, there seems to be no easy and natural way to show type and semantics preservation of this process. There are also some other methods such as [10], but we do not know any attempt to establish a framework for reasoning about register allocation process.

* This work was partially supported by Grant-in-aid for scientific research on priority area "informatics" A01-08, grant no:14019403.

P. Degano (Ed.): ESOP 2003, LNCS 2618, pp. 399–413, 2003.

The goal of this work is to establish a novel framework for reasoning about register allocation process and also for developing a practical register allocation algorithm.

Our strategy is to present register allocation as a series of proof transformations among proof systems that represents code languages with different variable usage. In an earlier work [8], the author has shown that a low-level code language can be regarded as a sequent-style proof system. In that work, a proof system deduces typing properties of a code. However, it is also possible to regard each "live range" of a variable as a type, and to develop a proof system to deduce properties of variable usage of a given code. Such a proof system must admit *structural rules*, e.g. those of *contraction*, *weakening* and *exchange*, to rearrange assumptions. The key idea underlying the present work is to regard those structural rules as register manipulation instructions and to represent a register allocation process as a proof transformation from a proof system with implicit structural rules to one with explicit structural rules. In this paradigm, liveness analysis is done by proof reconstruction similarly to type inference. Different from ordinary type inference, however, it always succeeds for any code and returns a proof, which is the code annotated with variable liveness information. The reconstructed proof is then transformed to another proof where allocation and deallocation of registers, and memory-register moves are explicitly inserted. The target machine code is extracted mechanically from the transformed proof.

Based on this idea, we have worked out the details of proof transformations for all the stages of register allocation, and have developed a register allocation algorithm. The correctness of the algorithm is an obvious corollary of this construction itself. Since structural rules only rearrange assumptions and do not change the computational meaning of a program, the resulting proof is equivalent to the original proof representing the given source code. Moreover, as being a proof system, our framework can be easily combined with a static type system of low-level code. Compared with the conventional approaches based on graph coloring, our framework is more general in that it uniformly integrate liveness analysis and register-memory moves.

We believe that the framework can be used to develop a practical register allocation algorithm. In order to demonstrate its practical feasibility, we have implemented the proposed method. Although the current prototype is a "toy implementation" and does not incorporated any heuristics, our limited experimentation confirms the effectiveness of our framework.

The major source of our inspiration is various studies on proof systems in substructural logic [9] and in linear logic [5]. They have attracted much attention as logical foundations for "resource management". To the author's knowledge, however, there does not seem to exist any attempt to develop a register allocation method using proof theoretical or type-theoretical frameworks.

The rest of the paper is organized as follows. Section 2 defines a proof system for a simple source language and gives a proof reconstruction algorithm. Section 3 gives a proof normalization algorithm to optimize live ranges of variables. Section 4 presents a proof transformation to a language with a fixed number of registers, and gives an algorithm to assign register numbers. Section 5 discusses some properties of the method and concludes the paper.

2 Proof System for Code Language and Liveness Analysis

To present our method, we define a simple code language. Let $x, y, \ldots$ range over a given countably infinite set of variables and c range over a given set of atomic constants. We consider the following instructions (ranged over by I), basic blocks (ranged over by B), and programs (ranged over by P).

$$\begin{aligned} I &::= x \texttt{ = } y \mid x \texttt{ = } c \mid x \texttt{ = } y \texttt{ + } z \mid \texttt{if } x \texttt{ goto } l \\ B &::= \texttt{return} x \mid \texttt{goto } l \mid I; B \\ P &::= \{l : B, \ldots, l : B\} \end{aligned}$$

There is no difficulty of adding various primitive operations other than +. It is also a routine practice to transform a conventional intermediate language into this representation by introducing necessary labels.

We base our development on a proof-theoretical interpretation of low-level code [8] where each instruction I is interpreted as an inference rule of the form

$$\frac{\Gamma' \rhd B \,:\, \tau}{\Gamma \rhd I; B \,:\, \tau}$$

indicating the fact that I changes machine state Γ to Γ' and continues execution of the block B. Note that a rule forms a bigger code from a smaller one, so the direction of execution is from the bottom to the top. If the above rule is the last inference step, then I is the first instruction to execute. The "return" instruction corresponds to an *initial sequent* (an axiom in the proof system) of the form

$$\Gamma, x : \tau \rhd \texttt{return } x \,:\, \tau$$

which returns the value of r to the caller. All the sequents in the same proof has the same result type determined by this rule.

Under this interpretation, each basic block becomes a proof in a sequent style proof system. A branching instruction is interpreted as a meta-level rule referring to an existing proof. For this purpose, we introduce a *label environment* (ranged over by $\mathcal{L}$) of the form $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\}$ specifying the type of each label, and define a proof system relative to a given label environment. We regard $\mathcal{L}$ as a function and write $\mathcal{L}(l_i)$ for the l_i's entry in $\mathcal{L}$.

To apply this framework to register allocation, we make the following two refinements. First, we regard a type not as a property of values (such as being an integer) but as a property of variable usage, and introduce a *type variable* for each *live range* of a variable. Occurrences of the same variable with different type variables imply that the variable has multiple live ranges due to multiple assignments. Second, we regard *structural rules* in sequent style proof system as (pseudo) instructions for allocation and de-allocation of variables (registers). The left-weakening rule corresponds (in the sense of Curry-Howard isomorphism) to the rule for discarding a register:

$$\frac{\Gamma \rhd \tau_0}{\Gamma, \tau \rhd \tau_0} \quad \Longrightarrow \quad \frac{\Gamma, x : \texttt{nil} \rhd B \,:\, \tau_0}{\Gamma, x : \tau \rhd \texttt{discard } x; B \,:\, \tau_0}$$

$$\Gamma, x : t \rhd \texttt{return}\ x\ :\ t \qquad \frac{\Gamma, x : \texttt{nil} \rhd B\ :\ t_0}{\Gamma, x : t \rhd \texttt{discard}\ x; B\ :\ t_0} \qquad \frac{\Gamma, x : t \rhd B\ :\ t_0}{\Gamma, x : \texttt{nil} \rhd x\ \texttt{=}\ c; B\ :\ t_0}$$

$$\frac{\Gamma, x : t_1, y : t_2 \rhd B\ :\ t_0}{\Gamma, x : \texttt{nil}, y : t_2 \rhd x\ \texttt{=}\ y; B\ :\ t_0} \qquad \frac{\Gamma, x : t_1, y : t_2, z : t_3 \rhd B\ :\ t_0}{\Gamma, x : \texttt{nil}, y : t_2, z : t_3 \rhd x\ \texttt{=}\ y\ \texttt{+}\ z; B\ :\ t_0}$$

$$\frac{\Gamma, x : t_1, y : t_2 \rhd B\ :\ t_0}{\Gamma, x : t_3, y : t_2 \rhd x\ \texttt{=}\ x\ \texttt{+}\ y; B\ :\ t_0}\ \text{(similarly for } x\ \texttt{=}\ y\ \texttt{+}\ x\text{)}$$

$$\frac{\Gamma, x : t \rhd B\ :\ t_0}{\Gamma, x : t \rhd \texttt{if}\ x\ \texttt{goto}\ l; B\ :\ t_0}\ (\text{if } \mathcal{L}(l) = \Gamma' \rhd t_0 \text{ such that } \Gamma' \subseteq \Gamma, x : t.)$$

$$\Gamma \rhd \texttt{goto}\ l\ :\ t\ (\text{if } \mathcal{L}(l) = \Gamma' \rhd t \text{ such that } \Gamma' \subseteq \Gamma.)$$

Fig. 1. $\mathcal{SSC}(L)$: proof system for liveness information

where $x : \texttt{nil}$ indicates that x is not live at this point. Assuming that τ is a true formula (inhabited type), the following valid variant of the left-contraction rule corresponds to the rule for allocating a new register.

$$\frac{\Gamma, \tau \rhd \tau_0}{\Gamma, \rhd \tau_0} \quad \Longrightarrow \quad \frac{\Gamma, x : \tau \rhd B\ :\ \tau_0}{\Gamma, x : \texttt{nil} \rhd \texttt{alloc}\ x; B\ :\ \tau_0}$$

Later, we shall see that exchange rules represent register-memory moves.

We let t range over type variables. A *type* τ is either t or `nil` (which is introduced to make type inference easier.) A *context* Γ is a mapping from a finite set of variables to types. For contexts Γ and Γ', we write $\Gamma \subseteq \Gamma'$ if Γ is included in Γ' as sets ignoring entries of the form $x : \texttt{nil}$. Fig. 1 gives the proof system for liveness. This is relative to a given label environment $\mathcal{L}$. A program $\{l_1 : B_1, \ldots, l_n : B_n\}$ is derivable under $\mathcal{L}$, if $\mathcal{L}(l_i) = \Gamma_i \rhd \tau_i$ and $\Gamma_i \rhd B_i\ :\ \tau_i$ for each $1 \leq i \leq n$. We call this proof system $\mathcal{SSC}(L)$[1]. We note that, in this definition, `alloc` is implicitly included in the rules for assignment. Furthermore, if the target variable of an assignment is one of its operands, then the assignment rule also includes `discard`. For example, an inference step for `x = x + y` discards the old usage of variable `x` and allocates a new usage for `x`. This is reflected by the different type variables for `x` in the rule.

We develop a proof reconstruction algorithm. For this purpose, we introduce *context variables* (denoted by ρ) and extend the set of contexts as follows.

$$\gamma ::= \Gamma \mid \rho \cdot \Gamma$$

For this set of terms, we can define a unification algorithm. We say that a set of context equations (i.e. a set of pairs of contexts) E is *well formed* if whenever $\rho \cdot \Gamma_1$ and $\rho \cdot \Gamma_2$ appear in E, $dom(\Gamma_1) = dom(\Gamma_2)$. Well-formedness is preserved by unification, and therefore it is sufficient to consider well formed equations. This

[1] The proof system for low-level code in [8] is called the *sequential sequent calculus*; hence the name. Also note that a program in general forms a cyclic graph, and therefore as a logical system it is inconsistent. It should be regarded as a type system of a recursive program, but we continue to use the term *proof system*.

property allows us to define a unification algorithm similarly to the standard unification. We note that although context terms are similar to record types, any extra machinery for record unification is not required. We can show the following.

Theorem 1. *There is an algorithm* CUNIF *such that for any set of well formed context equations E, if* $\text{CUNIF}(E) = S$ *then S is a unifier of E, and if there is a unifier S of E then* $\text{CUNIF}(E) = S'$ *such that $S = S'' \circ S'$ for some S''.*

We omit a simple definition of CUNIF and its correctness proof.

Using CUNIF and a standard unification algorithm UNIFY on types, we define a proof inference algorithm INFER. To present the algorithm, we first define some notations. We let Δ range over proofs. In writing a proof tree, we only include, at each inference step, the instruction that is introduced. We write $\dfrac{\Delta}{(\Gamma \rhd I : \tau)}$ if Δ is a proof whose end sequent is $\Gamma \rhd I : \tau$. If γ is a context containing x, $\gamma\{x : \tau\}$ is the context obtained from γ by replacing the value of x with τ. An *entry constraint* is a sentence of the form $l \ll \gamma \rhd \tau$ indicating the requirement that the block labeled with l must be a proof $\gamma' \rhd \tau$ such that $\gamma' \subseteq \gamma$. Let $\mathcal{L}$ be a label environment and $\mathcal{C}$ be a set of entry constraints. $\mathcal{L}(\mathcal{C})$ is the set of sentence of the form $\gamma' \rhd \tau' \ll \gamma \rhd \tau$ obtained from $\mathcal{C}$ by replacing each l appearing in $\mathcal{C}$ with $\mathcal{L}(l)$. We write $\overline{x}$ and $\overline{x : \tau}$ for a sequence of variables and a sequence of typed variables.

The algorithm INFER is given in Fig. 2. It takes a labeled set of basic blocks $\{l_1 : B_1, \ldots, l_n : B_n\}$ and returns a labeled set of proofs $\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}$. It first uses INFBLK to infer for each B_i its proof scheme (i.e. a proof containing context variables) together with a set of entry constraints. INFBLK proceeds by induction on the structure of B_i, i.e. it traverses B_i backward from the last instruction (`return` or `goto`). When it encounters a new variable, it introduces a fresh type variable for a new live range of the variable. When it encounters an assignment to x, it inserts `discard` x, and changes the type of x to `nil`, and continues toward the entry point of the code block. It generates an entry constraint for l for each branching instruction (`goto` l or `if` x `goto` l.) After having inferred proof schemes for blocks, the main algorithm gathers the set $\mathcal{L}(\mathcal{C})$ of constraints of the form $\gamma' \rhd t' \ll \gamma \rhd t$. The set of constraints is then solved by fixed point computation, where each iteration step picks one sentence $\rho' \cdot \Gamma' \rhd t' \ll \rho \cdot \Gamma \rhd t$ such that $\Gamma' \not\subseteq \Gamma$ or $t \neq t'$, and generates a minimal substitution S such that $S(\rho \cdot \Gamma) = \rho'' \cdot \Gamma''$, $S(\Gamma') \subseteq \Gamma''$ and $S(t') = S(t)$. Finally, the main algorithm INFER instantiates all the context variables with empty set to obtain a ground proof.

We establish the soundness of this algorithm. Let $\mathcal{C}$ be a set of entry constraints of $\mathcal{L}$; let $dom(\mathcal{C})$ be the set of labels mentioned in $\mathcal{C}$. We say that a substitution S is a solution of $\mathcal{C}$ under $\mathcal{L}$ if $dom(\mathcal{C}) \subseteq dom(\mathcal{L})$ and, for each constraint $l \ll \gamma \rhd \tau$ in $\mathcal{C}$, $\mathcal{L}(l) = \gamma' \rhd \tau'$, $S(\gamma') \subseteq S(\gamma)$, and $S(\tau') = S(\tau)$. From this definition, if S is a solution of $\mathcal{C}$ under $\mathcal{L}$, then $S(\mathcal{L})$ satisfies $S(\mathcal{C})$. We can show the following lemmas.

Lemma 1. *Let B be a code block. If* $\text{INFBLK}(B) = (\mathcal{C}, \Delta)$ *then for any solution $(\mathcal{L}, S)$ of $\mathcal{C}$, $S(\Delta)$ is a derivable proof under $\mathcal{L}$.*

$\textsc{InfBlk}(\texttt{return}\ x) = (\emptyset, \rho \cdot x : t \rhd \texttt{return}\ x : t) \quad (t, \rho \text{ fresh})$

$\textsc{InfBlk}(\texttt{goto}\ l) = (\{l \ll \rho \rhd t\}, \rho \rhd \texttt{goto}\ l : t) \quad (t, \rho \text{ fresh})$

$\textsc{InfBlk}(x\ \texttt{=}\ v; B) =$

$\quad \text{let } (\mathcal{C}_1, \dfrac{\Delta_0}{(\gamma_0 \rhd I_0 : t_0)}) = \textsc{InfBlk}(B)$

$\qquad S = \textsc{cunif}(\gamma_0, \rho_1 \cdot \overline{y : t_2}) \quad (\overline{y} = FV(v) \cup \{x\}, \text{and } \rho_1, \overline{t_2} \text{ fresh})$

$\qquad \{y_1, \ldots, y_k\} = \{y' | (y' : \texttt{nil}) \in S(\gamma_0), y' \in \overline{y}\}$

$\qquad \dfrac{\Delta_1}{(\gamma_1 \rhd \tau_1)} = S(\Delta_0)$

$\qquad \dfrac{\Delta_{i+1}}{(\gamma_{i+1} \rhd \tau_{i+1})} = \dfrac{\Delta_i}{\gamma_i\{y_i : t_i'\} \rhd \texttt{discard}\ y_i : \tau_i} \quad (t_i' \text{ fresh for each } 1 \le i \le k)$

$\quad \text{in if } x \in FV(v) \text{ then } (S(\mathcal{C}_1), \dfrac{\Delta_{k+1}}{\gamma_{k+1}\{x : t_{k+1}\} \rhd x\texttt{=}v : S(t_0)}) \quad (t_{k+1} \text{ fresh})$

$\qquad \text{else } \ (S(\mathcal{C}_1), \dfrac{\Delta_{k+1}}{\gamma_{k+1}\{x : \texttt{nil}\} \rhd x\texttt{=}v : S(t_0)})$

$\textsc{Solve}(\mathcal{C}) =$

$\quad \text{if there is some } (\rho_1 \cdot \Gamma_1 \rhd \tau_1 \ll \rho_2 \cdot \Gamma_2 \rhd \tau_2) \in \mathcal{C} \text{ such that } \Gamma_1 \not\subseteq \Gamma_2 \text{ or } \tau_1 \neq \tau_2 \text{ then}$

$\qquad \text{let } S_1 = \textsc{unify}(\{(\Gamma_1(x), \Gamma_2(x)) | x \in dom(\Gamma_1) \cap dom(\Gamma_1), \Gamma_1(x) \neq \texttt{nil}\} \cup \{(\tau_1, \tau_2))\})$

$\qquad\quad \Gamma_3 = \{x : \Gamma_1(x) | x \in (dom(\Gamma_1) \setminus dom(\Gamma_2)), \Gamma_1(x) \neq \texttt{nil}\}$

$\qquad\quad S_2 = [\rho_3 \cdot S_1(\Gamma_3)/\rho_2] \cup S_1 \quad (\rho_3 \text{ fresh})$

$\qquad \text{in } \textsc{Solve}(S_2(\mathcal{C})) \circ S_2$

$\quad \text{else } \emptyset$

$\textsc{Infer}(\{l_1 : B_1, \ldots, l_n : B_n\}) =$

$\quad \text{let } (\mathcal{C}_i, \dfrac{\Delta_i}{(\Gamma_i \rhd \tau_i)}) = \textsc{InfBlk}(B_i) \quad (1 \le i \le n)$

$\qquad (\mathcal{C}, \{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}) = (\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n, \{l_1 : \Delta_1, \ldots, l_n : \Delta_n\})$

$\qquad \mathcal{L} = \{l_i : \gamma_i \rhd \tau_i | \Delta_i \text{'s end sequent is of the form } \gamma_i \rhd B_i : \tau_i\}$

$\qquad S = \textsc{Solve}(\mathcal{L}(\mathcal{C}))$

$\qquad P = S(\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\})$

$\qquad \{\rho_1, \ldots, \rho_k\} = FreeContextVars(P)$

$\quad \text{in } [\emptyset/\rho_1, \ldots, \emptyset/\rho_k](P)$

Fig. 2. Some of the cases of proof reconstruction algorithm

Lemma 2. *If* $\textsc{Solve}(\mathcal{L}(\mathcal{C})) = S$ *and* $\{\rho_1, \ldots, \rho_k\}$ *is the set of free context variables in* $S(\mathcal{L}(\mathcal{C}))$ *then* $[\emptyset/\rho_1, \ldots, \emptyset/\rho_k] \circ S$ *is a solution of* $\mathcal{C}$ *under* $\mathcal{L}$.

Using these lemmas, we can show the following soundness theorem.

Theorem 2. *If* $\textsc{Infer}(\{\cdots, l_i : B_i \cdots\}) = \{\cdots, l_i : \dfrac{\Delta_i}{(\Gamma_i \rhd I_i : \tau_i)}, \cdots\}$ *then each* Δ_i *is a proof of* B_i *under the label environment* $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\}$, *and therefore the program* $\{l_1 : B_1, \ldots, l_n : B_n\}$ *is derivable under* $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\}$.

(1) An example source code

```
        i = 1
        s = 0
loop    c = i > n
        if c goto finish
        s = s + i
        i = i + 1
        goto loop
finish  return s
```

(2) The source program obtained by decomposing the source code into basic blocks

$\{l_1$: `i = 1; s = 0; goto` l_2,
l_2: `c = i > n; if c goto` l_4`; goto` l_3,
l_3: `s = s + i; i = i + 1; goto` l_2,
l_4: `return s`$\}$

(3) The inferred proof schemes of blocks, and the associated constraints

$$l_1:\ \dfrac{\dfrac{\rho_1\{\mathtt{i}:t_2,\mathtt{s}:t_3\}\rhd \mathtt{goto}\ l_2\ :\ t_1}{\rho_1\{\mathtt{i}:t_2,\mathtt{s}:\mathtt{nil}\}\rhd \mathtt{s{=}0}\ :\ t_1}}{\rho_1\{\mathtt{i}:\mathtt{nil},\mathtt{s}:\mathtt{nil}\}\rhd \mathtt{i{=}1}\ :\ t_1} \qquad l_3:\ \dfrac{\dfrac{\rho_3\{\mathtt{i}:t_9,\mathtt{s}:t_{10}\}\rhd \mathtt{goto}\ l_2\ :\ t_8}{\rho_3\{\mathtt{i}:t_{11},\mathtt{s}:t_{10}\}\rhd \mathtt{i{=}i{+}1}\ :\ t_8}}{\rho_3\{\mathtt{i}:t_{11},\mathtt{s}:t_{12}\}\rhd \mathtt{s{=}s{+}i}\ :\ t_8}$$

$$l_2:\ \dfrac{\dfrac{\rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\}\rhd \mathtt{goto}\ l_3\ :\ t_4}{\rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\}\rhd \mathtt{if\ c\ goto}\ l_4\ :\ t_4}}{\rho_2\{\mathtt{c}:\mathtt{nil},\mathtt{i}:t_6,\mathtt{n}:t_7\}\rhd \mathtt{c{=}i{>}n}\ :\ t_4} \qquad l_4:\ \ \rho_4\{\mathtt{s}:t_{13}\}\rhd \mathtt{return\ s}\ :\ t_{13}$$

$$l_2 \ll \rho_1\{\mathtt{i}:t_2,\mathtt{s}:t_3\}\rhd t_1 \qquad l_2 \ll \rho_3\{\mathtt{i}:t_9,\mathtt{s}:t_{10}\}\rhd t_8$$
$$l_3 \ll \rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\}\rhd t_4 \qquad l_4 \ll \rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\}\rhd t_4$$

(4) The reconstructed liveness proof of the program after constraint solving

$$l_1:\ \dfrac{\dfrac{\{\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{goto}\ l_2\ :\ t_1}{\{\mathtt{i}:t_2,\mathtt{n}:t_3\}\rhd \mathtt{s{=}0}\ :\ t_1}}{\{\mathtt{n}:t_3\}\rhd \mathtt{i{=}1}\ :\ t_1} \qquad l_3:\ \dfrac{\dfrac{\{\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{goto}\ l_2\ :\ t_1}{\{\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{i{=}i{+}1}\ :\ t_1}}{\{\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{s{=}s{+}i}\ :\ t_1}$$

$$l_2:\ \dfrac{\dfrac{\{\mathtt{c}:t_4,\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{goto}\ l_3\ :\ t_1}{\{\mathtt{c}:t_4,\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{if\ c\ goto}\ l_4\ :\ t_1}}{\{\mathtt{i}:t_2,\mathtt{n}:t_3,\mathtt{s}:t_1\}\rhd \mathtt{c{=}i{>}n}\ :\ t_1} \qquad l_4:\ \ \{\mathtt{s}:t_1\}\rhd \mathtt{return\ s}\ :\ t_1$$

Fig. 3. Example code and the reconstructed liveness proof

Fig. 3 shows an example of proof reconstruction. It lists (1) a sample source code in an informal notation, (2) the source program obtained by decomposing the given source code into a set of basic blocks, (3) the inferred proof schemes of the basic blocks and the associated set of constraints, and (4) the reconstructed liveness proof after constraint resolution. In this final proof, empty entries of the form x : `nil` are eliminated since they are no longer needed after the proof reconstruction have been completed.

We will use the sample source code (1) in Fig. 3 as the running example and will show examples of the proof transformation steps that follow (4) in Fig. 3. The examples above and those shown later are (reformatted) actual outputs of our prototype system.

3 Optimizing Liveness by Inserting Weakening Rule

The above proof inference algorithm always succeeds and returns a liveness proof for any given code. The resulting proof is, however, not the only possible one. The next step is to optimize the inferred proof by proof normalization.

The proof system has the freedom in terms of the places where `discard` is inserted. Our proof inference algorithm does not insert weakening rules (`discard` instructions) except for those required by assignments. All the necessary weakening rules are implicitly included in `goto` and `return` instructions. The optimization step is to introduce weakening rules explicitly and to move them toward the root of the proof tree (i.e. toward the entry point of the code) so that variables are discarded as early as possible. This is characterized as a proof normalization process with respect to the following commutative conversion (used in the specified direction) of proofs:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : \mathtt{nil} \rhd B \ : \ \tau_0}}{\Gamma, x : \tau \rhd \mathtt{discard}\ x; B \ : \ \tau_0}}{\Gamma, x : \tau \rhd I; \mathtt{discard}\ x; B \ : \ \tau_0} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : \mathtt{nil} \rhd B \ : \ \tau_0}}{\Gamma, x : \mathtt{nil} \rhd I; B \ : \ \tau_0}}{\Gamma, x : \tau \rhd \mathtt{discard}\ x; I; B \ : \ \tau_0} \qquad (\text{if } x \notin I)$$

Fig. 4 gives the optimization algorithm WEAKEN. In these definitions, we used the notations $\Gamma|_V$ for the restriction of Γ to a set of variables V, and $\Gamma - V$ for the context obtained from Γ by removing the assumptions of variables in V.

We write ADDWEAKEN(Δ) for the proof obtained from Δ by adding all the `discard` instructions just before each branching instruction so that the proof conforms to the proof system where the axioms are restricted to the following:

$$\{x : t\} \rhd \mathtt{return}\ x \ : \ t. \qquad \Gamma \rhd \mathtt{goto}\ l \ : \ \tau \ (\text{if } \mathcal{L}(l) = \Gamma \rhd \tau)$$

We write $\Delta \xrightarrow{*} \Delta'$ if Δ' is obtained from Δ by repeated application of the conversion rule. We can show the following.

Theorem 3. *For any proof Δ, if* $\mathrm{WK}(\Delta) = \Delta'$ *then* ADDWEAKEN(Δ) $\xrightarrow{*} \Delta'$.

Fig. 5 shows the optimized proof of the proof inferred for our running example in Fig. 3. Since `discard` is a pseudo instruction and is not needed in subsequent development, the algorithm erases them after the optimization is completed. In Fig. 5, `discard` step is shown in parenthesis to indicate this fact.

Let us review the results so far obtained. The labeled set of proofs obtained from a given program (a labeled set of basic blocks) by the combination of proof inference (INFER) and optimization (WEAKEN) is a code annotated with liveness information at each instruction step. The annotated liveness information is at least as precise as the one obtained by the conventional method. This is seen by observing the following property. If Δ contains a inference step $\Gamma \rhd I \ : \ \tau$ then all the variables in Γ are live at I and the interference graph of P must contain a completely connected subgraph of the length of Γ. Significant additional benefit of our liveness analysis is that it is presented as a typing annotation to the original program. This enables us to change the set of the target variables for register allocation dynamically, to which we now turn.

$$\textsc{Weaken}(\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}) =$$
$$\quad \text{let } \mathcal{E} = \{l_1 : \textsc{EntryVars}(\Delta_1), \ldots, l_n : \textsc{EntryVars}(\Delta_n)\}$$
$$\qquad (V_i, \Delta_i') = \mathrm{WK}(\Delta_i) \quad (1 \le i \le n)$$
$$\quad \text{in } \{l_1 : \Delta_1', \ldots, l_n : \Delta_n'\}$$

$$\textsc{EntryVars}(\frac{\Delta}{(\Gamma \rhd I \ : \ \tau)}) = \{x | x \in dom(\Gamma), \Gamma(x) \neq \texttt{nil})\}$$

In the following definition, $\mathcal{E}$ is a global environment defined in the main algorithm.

$$\mathrm{WK}(\Gamma \rhd \texttt{return } x \ : \ t) = (dom(\Gamma) \setminus \{x\}, \{x : \Gamma(x)\} \rhd \texttt{return } x \ : \ t)$$

$$\mathrm{WK}(\Gamma \rhd \texttt{goto } l \ : \ \tau) = (dom(\Gamma) \setminus \mathcal{E}(l),\ \Gamma|_{\mathcal{E}(l)} \rhd \texttt{goto } l \ : \ \tau)$$

$$\mathrm{WK}(\frac{\Delta}{\Gamma \rhd x \texttt{ = } v \ : \ \tau}) =$$
$$\quad \text{let } (V, \frac{\Delta_0}{(\Gamma_0 \rhd I_0 \ : \ \tau)}) = \mathrm{WK}(\Delta)$$
$$\qquad \{x_1, \ldots, x_n\} = FV(v) \cap V$$
$$\qquad V' = V \setminus \{x_1, \ldots, x_n\}$$
$$\qquad \frac{\Delta_i}{(\Gamma_i \rhd \_ \ : \ \_)} = \frac{\Delta_{i-1}}{\Gamma_{i-1}\{x_i : \Gamma(x_i)\} \rhd \texttt{discard } x_i \ : \ \tau} \quad \text{for each } x_i \ (1 \le i \le n)$$
$$\quad \text{in } (V', \ \frac{\Delta_n}{\Gamma - V' \rhd x \texttt{ = } v \ : \ \tau})$$

$$\mathrm{WK}(\frac{\Delta}{\Gamma \rhd \texttt{discard } x \ : \ \tau}) = \text{let } (V, \Delta_0) = \mathrm{WK}(\Delta) \text{ in } (V \cup \{x\},\ \Delta_0)$$

Fig. 4. Some cases of weakening (discard pseudo instruction) insertion algorithm

$$l_1: \ \frac{\frac{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{goto } l_2 \ : \ t_1}{\{\texttt{i} : t_2, \texttt{n} : t_3\} \rhd \texttt{s=0} \ : \ t_1}}{\{\texttt{n} : t_3\} \rhd \texttt{i=1} \ : \ t_1} \qquad l_3: \ \frac{\frac{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{goto } l_2 \ : \ t_1}{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{i=i+1} \ : \ t_1}}{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{s=s+i} \ : \ t_1}$$

$$l_2: \ \frac{\frac{\frac{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{goto } l_3 \ : \ t_1}{(\{\texttt{c} : t_4, \texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{discard c} \ : \ t_1)}}{\{\texttt{c} : t_4, \texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{if c goto } l_4 \ : \ t_1}}{\{\texttt{i} : t_2, \texttt{n} : t_3, \texttt{s} : t_1\} \rhd \texttt{c=i>n} \ : \ t_1} \qquad l_4 : \ \{\texttt{s} : t_1\} \rhd \texttt{return s} \ : \ t_1$$

Fig. 5. The result of weakening insertion optimization for the example in Fig. 3

4 Assigning Registers

We have so far considered a language with unbounded number of variables. A conventional approach to register allocation selects a subset of variables for the target of register allocation, and "spills" the others out. The treatment of spilled variables require ad-hoc strategies. Our approach provides a systematic solution to this problem using the liveness annotated code itself. We consider the set of

registers as a "working set" of the live variables at each instruction step, and maintain this working set. For this purpose, we define a new proof system whose sequents are of the form

$$\Sigma \mid \Pi \rhd_k B : \tau.$$

where Π is a *register context* whose length is bounded by the number k of available registers, and Σ is a *memory context* of unbounded length. We assume that k is no less than the number of assumptions needed for each instructions. In our case, instructions have at most 2 operands and therefore $k \geq 2$. Each logical rule (instruction) can only access assumptions in Π. To assess Σ, we introduce rules for `load` x and `store` x to move assumptions between Σ and Π:

$$\frac{\Sigma \mid \Pi, x : t_1 \rhd_k B : t}{\Sigma, x : t_1 \mid \Pi \rhd_k \texttt{load}\ x; B : t_0} \qquad \frac{\Sigma, x : t_1 \mid \Pi \rhd_k B : t_0}{\Sigma \mid \Pi, x : t_1 \rhd_k \texttt{store}\ x; B : t_0}\ (\text{if } |\Pi| < k)$$

where $|\Pi|$ denotes the length of Π. The other rules do not change Σ and are the same as before. We call the new proof system $\mathcal{SSC}(LE, k)$.

In a proof-theoretical perspective, the previous proof system $\mathcal{SSC}(L)$ implicitly admits *unrestricted exchange* so that any assumptions in Γ is freely available, while the new proof system $\mathcal{SSC}(LE, k)$ requires explicit use of the exchange rules to access some part of the assumptions. The next step of our register allocation method is to transform a proof obtained in the previous step into a proof in this new system. Since each inference rule only uses no more than k assumptions, the following is obvious.

Proposition 1. *There is an algorithm* Exchange *such that, for any provable program P in $\mathcal{SSC}(L)$,* Exchange(k, P) *is a program provable in $\mathcal{SSC}(LE, k)$, and if we ignore the distinction between Σ and Π, and erase* `load` *and* `store`*, then it is equal to P.*

Exchange traverses the code block, and whenever it detects an instruction whose operands are not in Π, it exchanges the necessary operands in Σ with some variables in Π, which are selected according to some strategy. The algorithm is straightforward except for this strategy of selecting variables to be saved. With the existence of branches, developing an optimal strategy is a difficult problem. Our prototype system adopted a simple lookahead strategy: it selects one control flow and traverses the instructions (up to a fixed number) to form an ordered list of variables that are more likely to be used in near future, and select those variables that are not appearing in the beginning of this list. In practical, we need more sophisticated heuristics, which is outside of the scope of this paper.

Fig. 6 shows the result of exchange insertion against the optimized proof shown in Fig. 4.

The final stage of our development is to assign a register number to each type variable in Π at each instruction step. We do this by defining yet another proof system where a type variable in Π has an attribute of a register number (ranged over by p). The set of instructions in this final proof system is as follows.

$$\begin{aligned} I ::= {} & x\ \texttt{=}\ y \mid x\ \texttt{=}\ c \mid x\ \texttt{=}\ x\ \texttt{+}\ x \mid \texttt{if}\ x\ \texttt{goto}\ l \\ & \mid \texttt{load (}p\texttt{,}x\texttt{)} \mid \texttt{store (}p\texttt{,}x\texttt{)} \mid \texttt{move}\ x[p_i \to p_j] \end{aligned}$$

`load` (p, x) moves variable x from Σ to Π and loads register p with the content of x. `store` (p, x) is its converse. `move` $x[p_i \to p_j]$ is an auxiliary instruction

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3, \mathtt{s} : t_1\} \rhd_3 \mathtt{goto}\ l_2 \ :\ t_1}}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3\} \rhd_3 \mathtt{s{=}0} \ :\ t_1}}{\emptyset \mid \{\mathtt{n} : t_3\} \rhd_3 \mathtt{i{=}1} \ :\ t_1}}{l_1:}
\qquad
\dfrac{\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3, \mathtt{s} : t_1\} \rhd_3 \mathtt{goto}\ l_2 \ :\ t_1}}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3, \mathtt{s} : t_1\} \rhd_3 \mathtt{i{=}i{+}1} \ :\ t_1}}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3, \mathtt{s} : t_1\} \rhd_3 \mathtt{s{=}s{+}i} \ :\ t_1}}{l_3:\ \{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3\} \rhd_3 \mathtt{load\ s} \ :\ t_1}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3\} \rhd_3 \mathtt{goto}\ l_3 \ :\ t_1}}{\{\mathtt{s} : t_1\} \mid \{\mathtt{c} : t_4, \mathtt{i} : t_2, \mathtt{n} : t_3\} \rhd_3 \mathtt{if\ c\ goto}\ l_4 \ :\ t_1}}{\{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3\} \rhd_3 \mathtt{c{=}i{>}n} \ :\ t_1}}{\emptyset \mid \{\mathtt{i} : t_2, \mathtt{n} : t_3, \mathtt{s} : t_1\} \rhd_3 \mathtt{store\ s} \ :\ t_1}}{l_2:}
\qquad
\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{s} : t_1\} \rhd_3 \mathtt{return\ s} \ :\ t_1}}{l_4:\ \{\mathtt{s} : t_1\} \mid \emptyset \rhd_3 \mathtt{load\ s} \ :\ t_1}
$$

Fig. 6. Converting to the dual context calculus

that changes the register allocated to x, which corresponds to register-register copy instruction. The rules for `load`, `store` and `move` are given below.

$$\frac{\Sigma \mid \Pi, x : t[p_2] \rhd_k I \ :\ t_0}{\Sigma \mid \Pi, x : t[p_1] \rhd_k \mathtt{move}\ x[p_1 \to p_2] \ :\ t_0}\ (p_1 \notin \Pi) \qquad \frac{\Sigma \mid \Pi, x : t[p] \rhd_k I \ :\ t_0}{\Sigma, x : t \mid \Pi \rhd_k \mathtt{load}\ (p, x) \ :\ t_0}$$

$$\frac{\Sigma, x : t \mid \Pi \rhd_k I \ :\ t_0}{\Sigma \mid \Pi, x : t[p] \rhd_k \mathtt{store}\ (p, x) \ :\ t_0}\ (\text{if } |\Pi, x : t[p]| \leq k,\ p \notin \Pi)$$

The other rules are the same as those in $\mathcal{SSC}(LE, k)$ except that in Π, each type variable has distinct register number attribute p. We call this proof system $\mathcal{SSC}(LEA, k)$.

Within a block, proof transformation from $\mathcal{SSC}(LE, k)$ to $\mathcal{SSC}(LEA, k)$ is straightforwardly done by a simple tail recursive algorithm (starting from the entry point) that keeps track of the current register assignment of Π and a set of free registers, and updates them every time when Π is changed due to assignment, load or store instructions. Since the length of Π is bounded by k, it is always possible to assign registers. An extra work is needed to adjust register assignment before a branching instruction so that the assignment at the branching instruction agrees with that of the target block. If the target block is not yet processed, then we can simply set the current register assignment of Π as the initial assignment for the block. If an assignment has already been done for the target block and it does not agree on the current assignment, then we need to permutate some registers by inserting `move` instructions using one temporary register. If there is no free register, we have to save one and then load after the permutation. In our current prototype implementation, we adopt a simple strategy of trying to allocate the same register to the same liveness type whenever possible by caching the past allocation. Minimizing register-register moves at branching instructions requires certain amount of heuristics, which is left as future work.

It should be noted, however, that this problem is much simpler than the problem of combining independently colored basic blocks. Our liveness analysis and the subsequent decompositions of variables into Σ and Π are done globally, and therefore the set Π of register contexts are guaranteed to agree when control flows merge.

The remaining thing to be done is to extract machine code from a proof in $\mathcal{SSC}(LEA, k)$. We consider the following target machine code.

$$I ::= \texttt{return r}i \mid \texttt{goto } l \mid \texttt{r}i \texttt{ = } c \mid \texttt{r}i \texttt{ = r}j \mid \texttt{if r}i \texttt{ goto } l$$
$$\mid \texttt{store (r}i\texttt{,}x\texttt{)} \mid \texttt{load (r}i\texttt{,}x\texttt{)} \mid \texttt{r}i \texttt{ = r}j \texttt{ + r}k$$

$\texttt{r}i$ is the register identified by number i. $\texttt{store (r}i\texttt{,}x\texttt{)}$ stores the register $\texttt{r}i$ to the memory location named x. $\texttt{load (r}i\texttt{,}x\texttt{)}$ loads the register $\texttt{r}i$ with the content of the memory location x. Since in a proof of $\mathcal{SSC}(LEA, k)$, each occurrence of variable in its register context Π is associate with a register number, it is straightforward to extract the target machine code by simply traversing a proof. For example, for the proof

$$\dfrac{\dfrac{\Delta}{(\Sigma \mid \Pi, x : t_1[1], y : t_2[2] \rhd_k I \ : \ t_0)}}{\Sigma \mid \Pi, y : t_2[2] \rhd_k x \texttt{ = } y \ : \ t_0}$$

we emit instruction "$\texttt{r1 = r2}$" and then continue to emit code for the proof Δ.

Fig. 7 shows the proof in $\mathcal{SSC}(LEA, k)$ and the machine code for our running example.

5 Conclusions and Discussions

We have presented a proof-theoretical approach to register allocation. In our approach, liveness analysis is characterized as proof reconstruction in a sequent-style proof system where a formula (or a type) represents a "live range" of a variable at each instruction step in a given code. Register manipulation instructions such as loading and storing registers are interpreted as *structural rules* in the proof system. Register allocation process is then regarded as a proof transformation from a calculus with implicit structural rules to one with explicit structural rules. All these proof transformation processes are effectively done, yielding a register allocation algorithm. The algorithm has been implemented, which demonstrates the practical feasibility of the method.

This is the first step toward proof theoretical framework for register allocation; there remain number of issues to be investigated – detailed comparisons with other approaches, relationship to other aspects of code generation such as instruction scheduling, robust implementation and evaluation etc. Below we include some discussion and suggestions for further investigation.

Correctness and other formal properties. In our approach, register allocation is presented as a series of proof transformations among proof systems that differ in their treatment of *structural rules.* Since structural rules do not affect the meanings, it is an immediate consequence that our approach preserves the meaning of the code. Since our method is a form of type system, it can smoothly be integrated in a static type system of a code language. By regarding liveness types as attribute of the conventional notion of types, we immediately get a register allocation method for a typed code language. Type-preservation is shown trivially by erasing liveness and register attributes, and merging the memory

The proof with register number annotation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0], \mathtt{s} : t_1[r2]\} \rhd_3 \mathtt{goto}\ l_2 : t_1}}{\emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{s{=}0} : t_1}}{l_1:\ \emptyset \mid \{\mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{i{=}1} : t_1}}{\{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{goto}\ l_3 : t_1}}{\{\mathtt{s} : t_1\} \mid \{\mathtt{c} : t_4[r2], \mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{if\ c\ goto}\ l_4 : t_1}}{\{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{c{=}i{>}n} : t_1}}{l_2:\ \emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0], \mathtt{s} : t_1[r2]\} \rhd_3 \mathtt{store\ s} : t_1}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0], \mathtt{s} : t_1[r2]\} \rhd_3 \mathtt{goto}\ l_3 : t_1}}{\emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0], \mathtt{s} : t_1[r2]\} \rhd_3 \mathtt{i{=}i{+}1} : t_1}}{\emptyset \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0], \mathtt{s} : t_1[r2]\} \rhd_3 \mathtt{s{=}s{+}i} : t_1}}{l_3:\ \{\mathtt{s} : t_1\} \mid \{\mathtt{i} : t_2[r1], \mathtt{n} : t_3[r0]\} \rhd_3 \mathtt{load\ s} : t_1}$$

$$\dfrac{\dfrac{}{\emptyset \mid \{\mathtt{s} : t_1[r0]\} \rhd_3 \mathtt{return\ s} : t_1}}{l_4:\ \{\mathtt{s} : t_1\} \mid \emptyset \rhd_3 \mathtt{load\ s} : t_1}$$

The extracted machine code:

```
l1 :   r1 = 1                    l3:    load r2,s
       r2 = 0                           r2 = r2 + r1
       goto l2                          r1 = r1 + 1
                                        goto l2
l2:    store r2,s
       r2 = r1 > r0              l4:    load r0,s
       if r2 goto l4                    return r0
       goto l3
```

Fig. 7. Example of register number assignment and code emission

and register context of each sequent. We also believe that our method can be combined with other static verification systems for low-level code.

Expressiveness. Our formalism covers the entire process in register allocation, and as a formalism, it appears to be more powerful than the one underlying the conventional method based on graph coloring. We have seen that liveness analysis is as strong as the conventional method using an interference graph. Since our formalism transforms the liveness annotated code, it provides better treatment for register-memory move than the conventional notion of "spilling". Although we have not incorporated various heuristics in our prototype implementation, our initial experimentation using our prototype system found that our method properly deals with the example of a "diamond" interference graph discussed in literature [1], for which the conventional graph coloring based approach cannot find an optimal coloring. Fig. 8 shows one simple example.

(1) The source program

```
L:  a = d + c
    b = a + d
    c = a + b
    d = b + c
    goto L
```

(2) The optimized liveness proof:

$$\frac{\{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{goto L} : t_1}{\{\texttt{b} : t_4, \texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{discard b} : t_1}$$
$$\frac{\{\texttt{b} : t_4, \texttt{c} : t_3\} \rhd \texttt{d=b+c} : t_1}{\{\texttt{a} : t_5, \texttt{b} : t_4, \texttt{c} : t_3\} \rhd \texttt{discard a} : t_1}$$
$$\frac{\{\texttt{a} : t_5, \texttt{b} : t_4\} \rhd \texttt{c=a+b} : t_1}{\{\texttt{a} : t_5, \texttt{b} : t_4, \texttt{d} : t_2\} \rhd \texttt{discard d} : t_1}$$
$$\frac{\{\texttt{a} : t_5, \texttt{d} : t_2\} \rhd \texttt{b=a+d} : t_1}{\{\texttt{a} : t_5, \texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{discard c} : t_1}$$
$$\frac{}{\{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{a=b+d} : t_1}$$

(3) The proof without `discard`:

$$\frac{\{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{goto L} : t_1}{\{\texttt{b} : t_4, \texttt{c} : t_3\} \rhd \texttt{d=b+c} : t_1}$$
$$\frac{\{\texttt{a} : t_5, \texttt{b} : t_4\} \rhd \texttt{c=a+b} : t_1}{\{\texttt{a} : t_5, \texttt{d} : t_2\} \rhd \texttt{b=a+d} : t_1}$$
$$\frac{}{\{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd \texttt{a=b+d} : t_1}$$

(4) The proof in $\mathcal{SSC}(LE, k)$:

$$\frac{\emptyset \mid \{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd_2 \texttt{goto L} : t_1}{\emptyset \mid \{\texttt{b} : t_4, \texttt{c} : t_3\} \rhd_2 \texttt{d=b+c} : t_1}$$
$$\frac{\emptyset \mid \{\texttt{a} : t_5, \texttt{b} : t_4\} \rhd_2 \texttt{c=a+b} : t_1}{\emptyset \mid \{\texttt{a} : t_5, \texttt{d} : t_2\} \rhd_2 \texttt{b=a+d} : t_1}$$
$$\frac{}{\emptyset \mid \{\texttt{c} : t_3, \texttt{d} : t_2\} \rhd_2 \texttt{a=b+d} : t_1}$$

(5) The proof in $\mathcal{SSC}(LEA, k)$:

$$\frac{\emptyset \mid \{\texttt{c} : t_3[r0], \texttt{d} : t_2[r1]\} \rhd_2 \texttt{goto L} : t_1}{\emptyset \mid \{\texttt{b} : t_4[r1], \texttt{c} : t_3[r0]\} \rhd_2 \texttt{d=b+c} : t_1}$$
$$\frac{\emptyset \mid \{\texttt{a} : t_5[r0], \texttt{b} : t_4[r1]\} \rhd_2 \texttt{c=a+b} : t_1}{\emptyset \mid \{\texttt{a} : t_5[r0], \texttt{d} : t_2[r1]\} \rhd_2 \texttt{b=a+d} : t_1}$$
$$\frac{}{\emptyset \mid \{\texttt{c} : t_3[r0], \texttt{d} : t_2[r1]\} \rhd_2 \texttt{a=b+d} : t_1}$$

(6) The extracted machine code

```
L:  r0 = r1 + r0
    r1 = r0 + r1
    r0 = r0 + r1
    r1 = r1 + r0
    goto L
```

Fig. 8. Example for a code whose interference graph forms a "diamond"

Liveness analysis and SSA-style optimization. The main strength of our method is the representation of liveness as type system of code itself. For example, as far as liveness analysis is concerned, our system already contains the effect of SSA (static single assignment) transformation [4] without actually performing the transformation. The effect of renaming a variable at each assignment is achieved by allocating a fresh type variable. The effect of ϕ function at control flow merge is achieved by unification of the type variables assigned to the same variable in different blocks connected by a branch instruction. Thanks to these effects, our liveness analysis achieves the accuracy of those that perform SSA transformation without introducing the complication of ϕ functions. Moreover, we believe that this property also allows us to combine various techniques of SSA-based optimization in our approach. For example, since each live range has distinct type variables, it is easy to incorporate constant propagation or dead code elimination. The detailed study on the precise relationship with our type-

based approach and SSA transformation is beyond the scope of the current work, and we would like to report it elsewhere.

Acknowledgments. The author thanks Tomoyuki Matsumoto for his help in implementing the prototype system, and Sin-ya Katsumata for insightful discussion at early stage of this work while the author was in Kyoto University.

References

1. P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.
2. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. ACM Symposium on Compiler Construction*, pages 98–105, 1982.
3. G. J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.
4. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
5. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
6. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. ACM POPL Symposium*, 1998.
7. G. Necula and P. Lee. Proof-carrying code. In *Proc. ACM POPL Symposium*, pages 106–119, 1998.
8. A. Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proc. International Symposium on Functional and Logic Programming*, Springer LNCS 1722, pages 300–318, 1999.
9. H. Ono and Y. Komori. Logics without the contraction rule. *Journal of Symbolic Logic*, 50(1):169–201, 1985.
10. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.
11. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. ACM POPL Symposium*, pages 149–160, 1998.