Register Allocation by Proof Transformation*

Atsushi Ohori

Japan Advanced Institute of Science and Technology,
FAX: +81 761 51-1149; ohori@jaist.ac.jp

Abstract. This paper presents a proof-theoretical framework for register allocation 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 the one with restricted variable access. In our framework, the set of registers acts as the "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 the proof transformation process. Its correctness is a simple corollary of our construction; the resulting proof is equivalent to the proof of the original code modulo the treatment of structural rules. This yields a simple yet powerful register allocation algorithm. The algorithm has been implemented, demonstrating the feasibility of the framework[1].

DRAFT. Comments are welcome.

1 Introduction

Register allocation is a process to convert an intermediate language to another language that is closer to machine code. As such, it should ideally be presented as a language transformation system that preserves the meaning of a given program – both its static and dynamic semantics. Such a framework will not only yield a 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 complements recent results on verifying type safety of low-level code, e.g. [13, 6, 7, 9]. 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* [4, 3]. 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

* **A revised version to appear in Proc. ESOP Symposium, April 2003.**

[1] I made the implementation available at:
http://www.jaist.ac.jp/ ohori/regalloc.tar.gz.
If interested, please copy the tar file and consult the included README.txt file. Please note that it is only a prototype for our evaluation of the framework.

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 for this process. There are also some other methods such as [11], but we do not know any attempt to establish a framework for reasoning about register allocation process.

The goal of this work is to establish a novel framework for reasoning about register allocation, 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 of different variable usage. In an earlier work [9], 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 property of 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 the properties of variable usage of a given code. Such a proof system has to admit *structural rules*, e.g. contraction, weakening and exchange, for rearranging assumptions. The key idea underlying our development 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 the one with explicit structural rules. In this paradigm, liveness analysis is done by proof reconstruction similarly to type inference. Different from type inference, however, it always succeeds for any code and returns a proof, which is the code annotated with liveness information at each instruction step. The obtained 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 general 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 assumption sets and do not change the computational meaning of a program, the resulting proof is equivalent to the original proof representing a 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 is 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. We found one typical example discussed in literature for which our prototype system produces a better code.

The major source of our inspiration is various studies on proof systems in substructural logic [10] 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. In [1], it was suggested that a (variant of) lambda term can be annotated with register usage. Such annotation could be useful, but it does not provide any method for register allocation using the annotated information.

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. Appendix shows an example output of our prototype system.

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).

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

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 [9] 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 bottom to 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

$$\Gamma, r : \tau \rhd \mathsf{return}(r)\ :\ \tau$$

which returns the value of r to the caller. All the sequnets in a 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 *liveness type variable* for each *live range* of a variable. Different types of the same variable indicate different live ranges of the variables. Second, we regard *structural rules* in a proof system as (pseudo) instructions for allocation and de-allocation of variables (registers). The left-weakening rule represents discarding a register:

$$\text{(Weaken-L)}\ \frac{\Gamma \rhd \tau_0}{\Gamma, \tau \rhd \tau_0} \quad \Longrightarrow \quad \frac{\Gamma, x : \mathsf{nil} \rhd B\ :\ \tau_0}{\Gamma, x : \tau \rhd \mathsf{discard}\ x; B\ :\ \tau_0}$$

where $x : \mathsf{nil}$ indicates that x is not used 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.

$$\text{(ContractTrue-L)}\ \frac{\Gamma, \tau \rhd \tau_0}{\Gamma, \rhd \tau_0} \quad \Longrightarrow \quad \frac{\Gamma, x : \tau \rhd B\ :\ \tau_0}{\Gamma, x : \mathsf{nil} \rhd \mathsf{alloc}\ x; B\ :\ \tau_0}$$

We shall show in Section 4 that the exchange rule represents register-memory move.

We let t range over liveness type variables. A *liveness type* (ranged over by τ) is either t or nil (which is introduced to make type inference easier.) Figure 1 gives the proof system for liveness. This is relative to a given labeled 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 $\mathcal{SSC}(L)$[2]. 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 $\mathsf{discard}$. For example, an inference step for $\mathsf{x} = \mathsf{x} + \mathsf{y}$ discards the old usage of variable x and allocates the new usage for x. This is reflected by the different type variables for x in the rule.

We develop a proof reconstruction algorithm for a given code. For this purpose, in this section, we regard Γ as a *sequence*, and we say that Γ is *well formed* if the occurrences of variables in Γ are pairwise distinct. 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 (pairs of contexts) E is *well formed* if for any pair $\rho \cdot \Gamma_1$ and $\rho \cdot \Gamma_2$ appearing in E, Γ_1 and Γ_2 are both well formed and $dom(\Gamma_1) = dom(\Gamma_2)$. Well-formedness is preserved by unification, and therefore it is sufficient to consider

[2] The proof system for low-level code in [9] 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*.

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

$$\frac{\Gamma, x : t_1, y : t_2 \rhd B \ : \ \tau_0}{\Gamma, x : \mathsf{nil}, y : t_2 \rhd x = y; B \ : \ t_0} \qquad \frac{\Gamma, x : t_1, y : t_2, z : t_3 \rhd B \ : \ t_0}{\Gamma, x : \mathsf{nil}, y : t_2, z : t_3 \rhd x = y + z; B \ : \ t_0}$$

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

$$\frac{\Gamma, x : t \rhd B \ : \ t_0}{\Gamma, x : t \rhd \mathsf{if}\ x\ \mathsf{goto}\ l; B \ : \ t_0} \ (\text{if } \mathcal{L}(l) \ll \Gamma, x : t \rhd t_0) \qquad \Gamma \rhd \mathsf{goto}\ l \ : \ t \quad (\text{if } \mathcal{L}(l) \ll \Gamma \rhd t)$$

Fig. 1. $\mathcal{SSC}(L)$: Proof System for Liveness Information

well formed equations, for which we can 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 such as [12, 8] 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 terms, we develop a proof inference algorithm INFER. We let Δ range over proofs. In writing a proof tree, we only include, at each inference step, the instruction that is introduced (i.e. the first instruction of the block). We write $\dfrac{\Delta}{(\Gamma \rhd I \ : \ \tau)}$ if Δ is a proof whose end sequent is $\Gamma \rhd I \ : \ \tau$ (i.e. I is the instruction introduce by the last inference step.) INFER is defined as an algorithm which 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 infers for each B_i its proof scheme (i.e. a proof containing context variables) together with a set of *entry constraints* of the form $l \ll (\gamma \rhd \tau)$ indicating the requirement that the block labeled with l must be a proof of the form $\gamma' \rhd \tau$ such that $\gamma' \subseteq \gamma$ (as sets, ignoring entries of the form $x : \mathsf{nil}$). The algorithm proceeds by induction on the structure of B_i, i.e. it traverses B_i backward from the last instruction (return or goto statement). When the algorithm encounters a new variable, then it introduces a fresh type variable for a new live range of the variable. When it encounters an assignment to a variable x, it inserts discard x, and changes the type of x to a fresh type variable and continues toward the entry point of the code block. After having inferred proofs of blocks, the algorithm solves the entry constraints and then instantiates all the context variables with empty sequence to obtain a ground proof.

The algorithm is given in Figure 2. In these definitions, we have used the following notations. Let γ be a context containing x, we write $\gamma\{x := \tau\}$ for the

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

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

$\textsc{InfBlk}(\mathsf{if}\ x\ \mathsf{then}\ l; B) =$

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

$\qquad S = \textsc{cunif}((\gamma_0, \rho_1 \cdot \{x : t_1\})) \quad (\rho_1, t_1 \text{ fresh})$

$\qquad \dfrac{\Delta_1}{(\gamma_1 \rhd \tau_1)} = \text{if } x : \mathsf{nil} \in S(\gamma_0) \text{ then } \dfrac{S(\Delta_0)}{S(\gamma_0)\{x := t_2\} \rhd \mathsf{discard}\ x \ : \ S(t_0)} \ (t_2 \text{ fresh})$

$\qquad\qquad \text{else } S(\Delta_0)$

$\quad \text{in } (S(\mathcal{C}_1) \cup \{l \ll (\rho_1 \rhd \tau_1)\}, \dfrac{\Delta_1}{\gamma_1 \rhd \mathsf{if}\ x\ \mathsf{goto}\ l \ : \ \tau_1})$

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

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

$\qquad \overline{y} = FV(v) \cup \{x\}$

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

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

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

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

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

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

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

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

$\qquad \mathcal{C} = [(\Gamma_1 \rhd \tau_1)/l_1, \ldots, (\Gamma_n \rhd \tau_n)/l_n](\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n)$

$\quad \text{in } (\mathcal{C}, \{l_1 : \Delta_1, \ldots, l_n : \Delta_n\})$

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

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

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

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

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

$\qquad\quad S_3 = \textsc{Solve}(S_2 \circ S_1(\mathcal{C}))$

$\qquad \text{in } S_3 \circ S_2 \circ S_1$

$\quad \text{else } \emptyset$

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

$\quad \text{let } (\mathcal{C}, \{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}) = \textsc{InfProg}(\{l_1 : B_1, \ldots, l_n : B_n\})$

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

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

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

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

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

Fig. 2. Proof Reconstruction Algorithm

context γ' obtained from γ by replacing the value of x with τ. We also write $\overline{x}$ and $\overline{x : \tau}$ for a sequence of variables and a sequence of typed variables.

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 $\mathcal{C}$, $\mathcal{L}(l) = \gamma' \rhd \tau'$, $S(\gamma') \subseteq S(\gamma)$ (ignoring the entries of the form $x : \mathsf{nil}$), 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* $\textsc{InfBlk}(B) = (\mathcal{C}, \Delta)$ *then for any solution $(\mathcal{L}, S)$ of $\mathcal{C}$, $S(\Delta)$ is a derivable proof under $\mathcal{L}$.*

Lemma 2. *Let $\mathcal{C}$ be a set of entry constraints of $\mathcal{L}$. If* $\textsc{Solve}(\mathcal{C}, \mathcal{L}) = S$ *then S is a solution of $\mathcal{C}$ under $\mathcal{L}$. Conversely, if $\mathcal{C}$ has a solution S under $\mathcal{L}$ then* $\textsc{Solve}(\mathcal{C}, \mathcal{L})$ *succeeds with some S' such that $S = S'' \circ S'$ for some S''.*

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 $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\}$.*

The proof inference algorithm is also "complete" in the following weak sense: Infer always infers a derivable proof for any provable program. In fact, any code is provable in the liveness proof system and Infer always succeeds. The resulting proof is, however, not the only possible one for the given code. The next step in our proof-directed register allocation is to optimize the inferred proof by proof normalization.

3 Optimizing Liveness by Inserting Weakening Rule

The criteria of optimizing a proof is to minimize each live range identified by a liveness type variable. The proof system has the freedom in terms of the places where discard is inserted, and the optimization algorithm is characterized as a proof normalization process with respect to the following commutative conversion (used in the specified direction) of the weakening rule.

$$\dfrac{\dfrac{\Gamma, x : \mathsf{nil} \rhd B \ : \ \tau_0}{\Gamma, x : \tau \rhd \mathsf{discard}\ x; B \ : \ \tau_0}}{\Gamma, x : \tau \rhd I; \mathsf{discard}\ x; B \ : \ \tau_0} \implies \dfrac{\dfrac{\Gamma, x : \mathsf{nil} \rhd B \ : \ \tau_0}{\Gamma, x : \mathsf{nil} \rhd I; B \ : \ \tau_0}}{\Gamma, x : \tau \rhd \mathsf{discard}\ x; I; B \ : \ \tau_0} \text{ if } I \text{ doesn't use } x$$

The 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 entrance of the code) so that variables are

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

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

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

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

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

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

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

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

Fig. 3. Weakening (discard pseudo instruction) Insertion Algorithm

discarded as early as possible. Figure 3 gives the algorithm Weaken performing this process. 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 environment obtained from Γ by removing the assumptions of variables in V.

We write $\textsc{AddWeaken}(\Delta)$ 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:

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

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* $\textsc{AddWeaken}(\Delta) \xrightarrow{*} \Delta'$.

After the liveness inference is completed, the pseudo instruction $\mathsf{discard}\ x$ and the empty assumption entries of the form $x : \mathsf{nil}$ are no longer needed. So we erase them from a proof.

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 as precise as the one obtained by the conventional method. We can show the following.

Theorem 4. *Let P be a program and let $\{l_i : \Delta_i\}$ be a proof obtained from P by the proof inference followed by weakening insertion. If some Δ_i contains an assumption Γ of length more than k then P's interference graph cannot be k-colorable.*

This is proved by showing 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 according to the change of liveness, to which we now turn.

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 allocating, 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 an *active context* whose length is bounded by the number k of available registers, and Σ is a *saved context* of unbounded length. Each logical rule (instruction) can only access assumptions in Π. To assess Σ, the system introduces $\mathsf{load}\ x$ and $\mathsf{store}\ x$ to move assumptions between Σ and Π, whose rules are:

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

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 one in this proof system. Figure 4 gives this algorithm. In the algorithm, $\textsc{UseOrder}(dom(\Pi), \Delta)$ denotes the list of variables in $dom(\Pi)$ ordered according to the occurrence in Δ (those occurring near the end of Δ come earlier in the list). For this algorithm, the following property is easily shown.

Theorem 5. *For a provable program P in $\mathcal{SSC}(L)$, if*

$$\textsc{Exchange}(k, \textsc{Weaken}(P)) = P'$$

then 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 *in P', then it is equal to* $\textsc{Weaken}(P)$.

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

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

$\mathsf{load}\ (p, x)$ moves variable x from Σ to Π and loads register p with the content of x. $\mathsf{store}\ (p, x)$ is its converse. $\mathsf{move}\ x[p_i \rightarrow p_j]$ is an auxiliary instruction that changes the register allocated to x. The rules for load, store and move are:

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

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

The other rules are the same as in $\mathcal{SSC}(LE, k)$ except that in Π, each liveness 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

$\textsc{Exchange}(k, \{l_1 : \dfrac{\Delta_1}{(\Gamma_1 \rhd I_1 \,:\, t_1)}, \ldots, l_n : \dfrac{\Delta_n}{(\Gamma_n \rhd I_n \,:\, t_n)}\}) =$

let k_i be $k-1$ if l_i is a target of conditional branch "if x goto l_i" otherwise k
V_i be the first (at most) k_i elements in $\textsc{UseOrder}(FV(B_i), \Delta_i)$
Π_i be the restriction of Γ_i on V_i
Σ_i be $\Gamma_i \setminus \Pi_i$
$\mathcal{E} = \{l_1 : (\Sigma_1, \Pi_1), \ldots, l_n : (\Sigma_n, \Pi_n)\}$
in $\{l_1 : \mathrm{EX}(\Sigma_1, \Pi_1, \Delta_1), \ldots, l_n : \mathrm{EX}(\Sigma_n, \Pi_n, \Delta_n)\}$

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

$\mathrm{EX}(\Sigma, \Pi, \Gamma \rhd \mathsf{return}(x) \,:\, t) = (\Sigma \mid \Pi \rhd_k \mathsf{return}(x) \,:\, t)$

$\mathrm{EX}(\Sigma, \Pi, \Gamma \rhd \mathsf{goto}\ l \,:\, t) =$
let $(\Sigma_0, \Pi_0) = \mathcal{E}(l)$
$x_1, \ldots, x_m = dom(\Pi_0) \setminus dom(\Pi)$
$y_1, \ldots, y_n = dom(\Sigma_0) \setminus dom(\Sigma)$
$\Delta_0 = \Sigma_0 \mid \Pi_0 \rhd_k \mathsf{goto}\ l \,:\, t$
$\Delta_i = \dfrac{\Delta_{i-1}}{\Sigma_{i-1}\{x_i : \Pi_{i-1}(x_i)\} \mid \Pi_{i-1} - \{x_i\} \rhd_k \mathsf{load}\ x_i \,:\, t}\ (1 \le i \le m)$
$\Delta_{m+j} = \dfrac{\Delta_{m+j-1}}{\Sigma_{m+j-1} - \{y_j\} \mid \Pi_{m+j-1}\{y_j : \Pi_{m+j-1}(y_j)\} \rhd_k \mathsf{store}\ y_j \,:\, t}\ (1 \le j \le n)$
in Δ_{m+n}

$\mathrm{EX}(\Sigma, \Pi, \Gamma \rhd \mathsf{if}\ x\ \mathsf{goto}\ l \,:\, t) =$
let $(\Sigma', \Pi') = \mathcal{E}(l)$
$x_1, \ldots, x_m = (dom(\Pi') \cup \{x\}) \setminus dom(\Pi)$
$y_1, \ldots, y_n = dom(\Sigma') \setminus dom(\Sigma)$
$\Sigma_0 = (\Sigma - \{x_1, \ldots, x_m\})\{y_1 : \Pi(y_1), \ldots, y_n : \Pi(y_n)\}$
$\Pi_0 = (\Pi - \{y_1, \ldots, y_n\})\{x_1 : \Sigma(x_1), \ldots, x_n : \Sigma(x_m)\}$
$\Delta_0 = \dfrac{\mathrm{EX}(\Sigma_0, \Pi_0, \Delta)}{\Sigma_0 \mid \Pi_0 \rhd_k \mathsf{if}\ x\ \mathsf{goto}\ l \,:\, t}$
$\Delta_i = \dfrac{\Delta_{i-1}}{\Sigma_{i-1}\{x_i : \Pi_{i-1}(x_i)\} \mid \Pi_{i-1} - \{x_i\} \rhd_k \mathsf{load}\ x_i \,:\, t}\ (1 \le i \le m)$
$\Delta_{m+j} = \dfrac{\Delta_{m+j-1}}{\Sigma_{m+j-1} - \{y_j\} \mid \Pi_{m+j-1}\{y_j : \Pi_{m+j-1}(y_j)\} \rhd_k \mathsf{store}\ y_j \,:\, t}\ (1 \le j \le n)$
in Δ_{m+n}

$\mathrm{EX}(\Sigma, \Pi, \dfrac{\Delta}{\Gamma \rhd x = v \,:\, t}) =$
let $\{x_1, \ldots, x_m\} = FV(v) \setminus dom(\Pi)$
$n = m + (\text{length of } \Pi) - k$
$y_1, \ldots, y_n$ be the last n variables in $\textsc{UseOrder}(dom(\Pi) \setminus \{x_1, \ldots, x_m\}, \Delta)$
$\Sigma_0 = (\Sigma - \{x_1, \ldots, x_m\})\{y_1 : \Pi(y_1), \ldots, y_n : \Pi(y_n)\}$
$\Pi_0 = (\Pi - \{y_1, \ldots, y_n\})\{x_1 : \Sigma(x_1), \ldots, x_m : \Sigma(x_m)\}$
$\Sigma' = \text{if } x \in dom(\Pi) \text{ then } \Sigma_0 \text{ else } \Sigma_0 - \{x\}$
$\Pi' = \text{if } x \in dom(\Pi) \text{ then } \Pi_0 \text{ else } \Pi_0\{x : \Sigma(x)\}$
$\Delta_0 = \dfrac{\mathrm{EX}(\Sigma', \Pi', \Delta)}{\Sigma_0 \mid \Pi_0 \rhd_k x = v \,:\, t}$
$\Delta_i = \dfrac{\Delta_{i-1}}{\Sigma_{i-1}\{x_i : \Pi_{i-1}(x_i)\} \mid \Pi_{i-1} - \{x_i\} \rhd_k \mathsf{load}\ x_i \,:\, t}\ (1 \le i \le m)$
$\Delta_{m+j} = \dfrac{\Delta_{m+j-1}}{\Sigma_{m+j-1} - \{y_j\} \mid \Pi_{m+j-1}\{y_j : \Pi_{m+j-1}(y_j)\} \rhd_k \mathsf{store}\ y_j \,:\, t}\ (1 \le j \le n)$
in Δ_{m+n}

Fig. 4. Exchange (load and store instruction) Insertion Algorithm

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.

Minimizing register-register moves at branching instructions is a difficult problem and this issues is left as a future work. In our current prototype implementation, we adopt a simple heuristics of trying to allocate the same register to the same liveness type whenever possible by caching the past allocation.

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.

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

$\mathtt{r}i$ is the register identified by number i. store $(\mathsf{r}i,x)$ stores the register $\mathtt{r}i$ to the memory location named x. load $(\mathsf{r}i,x)$ loads the register $\mathtt{r}i$ with the content of the memory location x. Since in a proof of $\mathcal{SSC}(LEA, k)$, each occurrence of variable in its active 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

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

we emit instruction "`r1 = r2`" and then continue to emit code for the proof Δ.

5 Conclusions and 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 the one with explicit structural rules. All these proof transformation processes are effectively done, yielding a register allocation algorithm. The algorithm has been implemented, which confirms the practical feasibility of the method.

This is the first step toward proof theoretical framework for register allocation; detailed comparisons with other approaches, robust implementation, benchmarks and other evaluation etc. are beyond the scope of the present work. Below we include some discussion and our initial evaluation.

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 saved and active 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 shown 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 register "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 [2], for which the conventional graph coloring based approach cannot find an optimal coloring.

Optimization. The main strength of our method is the representation of liveness as type system of code itself. Moreover, our usage of type variables achieves the similar effect of SSA (static single assignment) transformation without actual SSA transformation. This allows us to combine various techniques of type-based and SSA-based optimization. For example, since each live range has distinct type variables, it is easily incorporate constant propagation or dead code elimination. The detailed study on this topic, especially on the relationship with SSA transformation, is beyond the scope of the current work and we would like to report them elsewhere.

Acknowledgments

The author thank Tomoyuki Matsumoto for helpful discussion and for his help in implementing the prototype system.

References

1. J. Agat. Types for register allocation. In *Proc. International Workshop on the Implementation of Functional Languages*, pages 92–111, 1998.
2. P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Tran. Prog. Lang. and Syst.*, 16(3):428–455, May 1994.
3. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proc. ACM Symposium on Compiler Construction*, pages 98–105, 1982.

4. 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.
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. A polymorphic record calculus and its compilation. *ACM Trans. Prog. Lang. and Syst.*, 17(6):844–895, 1995.
9. A. Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proc. FLOPS*, LNCS 1722, pp 300-318, 1999.
10. H. Ono and Y. Komori. Logics without the contraction rule. *Journal of Symbolic Logic*, 50(1):169–201, 1985.
11. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. and Syst.*, 21(5):895–913, 1999.
12. D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM POPL Symposium*, pages 242–249, 1989.
13. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. ACM POPL Symposium*, pages 149–160, 1998.

Appendix : An example computed in our prototype implementation

The following are actual outputs of our prototype system. Each proof is printed up-side-down so that the direction of execution is from top to bottom for better readability. In the following, $\mathsf{L}i$ is a liveness type variable and $\mathsf{ctx}(i)$ is a context variable.

(1) Source code:

```
       i = 1
       S = 0
loop   c = i > n
       if c goto finish
       i = i + 1
       S = S + i
       goto loop
finish return(S)
```

(2) The inferred liveness proof:

```
_start:
  ctx(10){S:_,  i:_}   |- i = 1       : L10
  ctx(10){S:_,  i:L12} |- S = 0       : L10
  ctx(10){S:L11,i:L12} |- GOTO loop   : L10
loop:
   ctx(7){c:_,  i:L9, n:L8}  |- c = i > n   : L6
   ctx(7){c:L7, i:L9, n:L8}  |- if c goto finish : L6
   ctx(7){c:L7, i:L9, n:L8}  |- GOTO _a     : L6
_a:
  ctx(3){S:L4, i:L5}  |- i = i + 1   : L1
  ctx(3){S:L4, i:L3}  |- S = S + i   : L1
  ctx(3){S:L2, i:L3}  |- GOTO loop   : L1
finish:
  ctx(0){S:L0}  |- RETURN(S)   : L0
with the constraints
  loop < ctx(3){S:L2, i:L3}  : L1
  loop < ctx(10){S:L11,i:L12} : L10
  _a  < ctx(7){c:L7, i:L9, n:L8}  : L6
  finish < ctx(7){c:L7, i:L9, n:L8}  : L6
```

(3) The proof obtained by solving constraints, instantiating context variables, and proof normalization:

```
_start:
                   {n:L8}  |- i = 1       : L11
              {i:L12,n:L8}  |- S = 0       : L11
       {S:L11,i:L12,n:L8}  |- GOTO loop   : L11
loop:
       {S:L11,i:L12,n:L8}  |- c = i > n   : L11
 {S:L11,c:L7, i:L12,n:L8}  |- if c goto finish : L11
       {S:L11,i:L12,n:L8}  |- GOTO _a     : L11
_a:
       {S:L11,i:L12,n:L8}  |- i = i + 1   : L11
       {S:L11,i:L12,n:L8}  |- S = S + i   : L11
       {S:L11,i:L12,n:L8}  |- GOTO loop   : L11
finish:
                   {S:L11} |- RETURN(S)   : L11
```

(4) The proof in the dual context calculus:

```
_start :
        {} & {n:L8}             |- i = 1       : r(11)
        {} & {i:L12,n:L8}       |- S = 0       : r(11)
        {} & {S:L11,i:L12,n:L8} |- GOTO loop   : r(11)
loop :
        {} & {S:L11,i:L12,n:L8} |- store S     : r(11)
   {S:L11} & {i:L12,n:L8}       |- c = i > n   : r(11)
   {S:L11} & {c:L7, i:L12,n:L8} |- if c goto finish : r(11)
   {S:L11} & {i:L12,n:L8}       |- GOTO _a     : r(11)
_a :
   {S:L11} & {i:L12,n:L8}       |- i = i + 1   : r(11)
   {S:L11} & {i:L12,n:L8}       |- load S      : r(11)
        {} & {S:L11,i:L12,n:L8} |- S = S + i   : r(11)
        {} & {S:L11,i:L12,n:L8} |- GOTO loop   : r(11)
finish :
   {S:L11} & {}                 |- load S      : r(11)
        {} & {S:L11}            |- RETURN(S)   : r(11)
```

(5) The proof with register number annotation:

```
_start :
      {} & {n:L8[r0]}                    |- i = 1       : L11
      {} & {i:L12[r1],n:L8[r0]}          |- S = 0       : L11
      {} & {S:L11[r2],i:L12[r1],n:L8[r0]} |- goto loop   : L11
loop :
      {} & {S:L11[r2],i:L12[r1],n:L8[r0]} |- store S     : L11
 {S:L11} & {i:L12[r1],n:L8[r0]}          |- c = i > n   : L11
 {S:L11} & {c:L7[r2], i:L12[r1],n:L8[r0]} |- if c goto finish : L11
 {S:L11} & {i:L12[r1],n:L8[r0]}          |- goto _a     : L11
_a :
 {S:L11} & {i:L12[r1],n:L8[r0]}          |- i = i + 1   : L11
 {S:L11} & {i:L12[r1],n:L8[r0]}          |- load S      : L11
      {} & {S:L11[r2],i:L12[r1],n:L8[r0]} |- S = S + i   : L11
      {} & {S:L11[r2],i:L12[r1],n:L8[r0]} |- goto loop   : L11
finish :
 {S:L11} & {}                            |- load S      : L11
      {} & {S:L11[r0]}                   |- return(S)   : L11
```

(6) The generated machine code:

```
   _start :                          _a :
          r1 = 1                            r1 = r1 + 1
          r2 = 0                            load r2,S
          goto loop                         r2 = r2 + r1
   loop :                                   goto loop
          store r2,S                 finish :
          r2 = r1 > r0                      load r0,S
          if r2 goto finish                 return(r0)
          goto _a
```