View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE @ DIRECT®

Science of Computer Programming 50 (2004) 161 – 187

Science of Computer Programming

www.elsevier.com/locate/scico

Register allocation by proof transformation

Atsushi Ohori[1]

School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa 923-1292, Japan

Received 5 July 2003; received in revised form 14 November 2003; accepted 11 December 2003

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". 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 serves as a basis for reasoning about formal properties of register allocation process, and it also yields a clean and systematic register allocation algorithm. The algorithm has been implemented, demonstrating the feasibility of the framework.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Register allocation; Liveness analysis; Proof transformation; Structural rules

1. Introduction

Register allocation is a process to convert an intermediate language to another language closer to machine code. It should ideally be presented as a language transformation system that preserves the meaning of a program—both its static and dynamic semantics. Such 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, which will complement recent results on verifying low-level code,

[1] Partially supported by Grant-in-aid for scientific research on priority area "informatics" A01-08, Grant No:14019403.

E-mail address: ohori@jaist.ac.jp (A. Ohori).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.01.005

e.g. [18,10,11,13]. If register allocation is formalized as meaning preserving program transformation, then one can factor out architecture-dependent register allocation process in code verification. Unfortunately, however, it appear to be difficult to establish such results for existing methods of register allocation.

The most widely used register allocation method 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 linear scan [16], but we do not know any attempt to establish a framework for reasoning about register allocation process.

The goal of this work is to develop a novel framework for reasoning about register allocation process, and for developing a practical register allocation algorithm. Our strategy is to present register allocation as a series of proof transformations among proof systems for code languages with different variable usage. In an earlier work [13], the author has shown that a low-level code language can be regarded as a sequent-style proof system in the sense of Curry–Howard isomorphism [4,9]. In that work, a proof system deduces typing properties of a code. However, it should also be 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. Proofs in such a proof system should contain enough information to perform register allocation.

A sequent-style 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 a 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. Fig. 1 gives the transformation steps of our 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 readily combined with a static type system of low-level code.

The primary motivation of this work is to provide a framework for reasoning about register allocation process. However, we also believe that the framework can be used to develop a practical register allocation algorithm. Compared with the conventional approaches based on graph coloring, our framework is more general in that it uniformly integrates liveness analysis and register–memory moves. This will allows us to

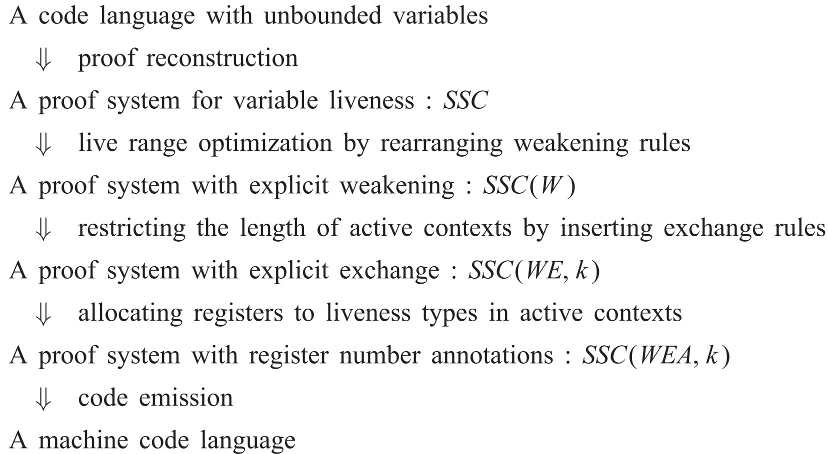

Fig. 1. The structure of register allocation by proof transformation.

develop a register allocation algorithm more systematically. In order to demonstrate its practical feasibility, we have implemented the proposed method. Although the current prototype is a "toy implementation" and does not incorporate 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 [15] and linear logic [6]. (See also [14,19] for tutorials on the related subjects.) 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 simple source language and develops a proof system *SSC* for variable liveness. Section 3 gives a proof reconstruction algorithm. Section 4 defines a proof system $SSC(W)$ with explicit weakening rules, and presents an optimization method for live ranges of variables by proof transformation from *SSC* to $SSC(W)$ and proof normalization in $SSC(W)$. Section 5 presents a proof transformation to a proof system $SSC(WE,k)$ for a language with a fixed number of active variables. Section 6 gives a proof transformation to a proof system $SSC(WEA,k)$ for machine code. Section 7 discusses some properties of the method and concludes the paper.

2. A proof system for variable liveness

To present our method, we define a simple code language. Let $x, y, \ldots$ range over a given countably infinite set of variables and let 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):

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

Here, we only include operation $+$. There is no difficulty in adding various other primitives. It is also a routine practice to transform a conventional intermediate language into this representation by introducing necessary labels.

A basic block ends with branching instructions. In this source language, conditional branch must be followed by a branch instruction to form a combined statement of the form "$\texttt{if}\ x\ \texttt{goto}\ l\texttt{;}\ \texttt{goto}\ l$". This restriction is needed for proof reconstruction. In the subsequent proof transformation we shall define in Section 4, we need to relax this restriction and consider the above as a sequence of two instructions so that we can insert a pseudo-instruction for discarding variable x if it is no longer live after this usage.

We base our development on a proof-theoretical interpretation of low-level code [13] 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 x to the caller. All the sequents in a 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 through a label. To account for this feature, we introduce a *label environment* (ranged over by $\mathscr{L}$) of the form $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\}$ specifying the assumption that a block of label l_i is a proof of the form $\Gamma_i \rhd B_i \ : \ \tau_i$, and define a proof system relative to a given label environment. We regard $\mathscr{L}$ as a function and write $\mathscr{L}(l_i)$ for the l_i's entry in $\mathscr{L}$.

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. A live range corresponds to a set of occurrences of a variable holding the same value generated by an assignment. Since we consider global register allocation, a live range in general spans multiple basic blocks. Occurrences of the same variable with different liveness type variables imply that the variable has multiple live

ranges due to multiple assignments. Second, we regard *structural rules* in a 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:

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

where $x : \mathtt{nil}$ 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.

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

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

We let t range over liveness 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, we used the following notations. $\Gamma|_V$ is the restriction of Γ to a set of variables V, and $\Gamma|_{\bar{V}}$ is the context obtained from Γ by removing the assumptions of variables in V. For contexts Γ and Γ', we write $\Gamma \subseteq \Gamma'$ if Γ is included in Γ' as sets ignoring entries of the form "$x : \mathtt{nil}$", i.e. for each $x \in dom(\Gamma)$ if $\Gamma(x) \neq \mathtt{nil}$ then $x \in dom(\Gamma')$ and $\Gamma'(x) = \Gamma(x)$.

Fig. 2 gives the proof system for liveness. This is relative to a given label environment $\mathscr{L}$. We call this proof system *SSC*.[2] We let Δ range over proofs of blocks. We write $\mathscr{L} \vdash_{SSC} \Delta$ if Δ is derivable in *SSC* under $\mathscr{L}$. A proof of a program is of the form $\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}$. Let the end-sequent of Δ_i be of the form $\Gamma_i \rhd B_i \,:\, \tau_i$. A proof of a program $\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}$ is derivable in *SSC*, written $\vdash_{SSC} \{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}$, if, for each $1 \leqslant i \leqslant n$, $\{l_1 : \Gamma_1 \rhd \tau_1, \ldots, l_n : \Gamma_n \rhd \tau_n\} \vdash_{SSC} \Delta_i$. This mechanism is the same as the typing rule for mutually recursive function definitions.[3]

In our informal explanation above, we have pointed out that (a variant) of left-contraction corresponds to `alloc` pseudo-instruction for allocating a new variable. In an actual code language, the only point where a new live range is introduced is an assignment statement. So, in the definition of *SSC* above, `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 upper sequent and the lower sequent of the rule.

---

[2] The proof system for low-level code in [13] is called the *sequential sequent calculus*; hence the name.

[3] 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*.

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

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

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

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

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

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

Fig. 2. A proof system for variable liveness: *SSC*.

3. Liveness analysis through proof reconstruction

The first step in register allocation is liveness analysis, which determines live ranges of variables. In our formalism, this is done by reconstructing a proof in *SSC* for a given raw code.

We develop a proof reconstruction algorithm by inferring a *proof scheme* using unification, similarly to type inference. For this purpose, we introduce *context variables* (denoted by ρ) and consider *context schemes* as follows:

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

We write $dom(\gamma)$ for the set of variables in γ, i.e., $dom(\rho \cdot \Gamma) = dom(\Gamma)$. We also write $\Gamma \cup \gamma$ to denote the context obtained from γ by adding Γ, i.e., if $\gamma = \Gamma'$ then $\Gamma \cup \gamma = \Gamma \cup \Gamma'$ and if $\gamma = \rho \cdot \Gamma'$ then $\Gamma \cup \gamma = \rho \cdot (\Gamma \cup \Gamma')$.

A *substitution* S is a finite function which maps context variables to contexts and type variables to types. A substitution S is extended to the function S' on the set of all type variables and context variables by setting $S'(t) = t$ $(t \notin dom(S))$ and $S'(\rho) = \rho$ $(\rho \notin dom(S))$. The result of applying a substitution to a context is defined as

$$S(\gamma) = \begin{cases} \{x : S'(\Gamma(x)) \mid x \in dom(\Gamma)\} & \text{if } \gamma = \Gamma, \\ S(\Gamma) \cup S'(\rho) & \text{if } \gamma = \rho \cdot \Gamma. \end{cases}$$

$$\text{CUNIFY}(\rho \cdot \Gamma_1, \rho \cdot \Gamma_2) = \text{UNIFY}(\{(\Gamma_1(x), \Gamma_2(x)) \mid x \in dom(\Gamma_1) \cap dom(\Gamma_2)\})$$

$$\begin{aligned}
&\text{CUNIFY}(\rho_1 \cdot \Gamma_1, \rho_2 \cdot \Gamma_2) = \\
&\quad \text{let } \Gamma_1' = \Gamma_2|_{\overline{dom(\Gamma_1)}} \\
&\quad\quad \Gamma_2' = \Gamma_1|_{\overline{dom(\Gamma_2)}} \\
&\quad\quad \rho \text{ fresh} \\
&\quad\quad S = \text{UNIFY}(\{(\Gamma_1(x), \Gamma_2(x)) \mid x \in dom(\Gamma_1) \cap dom(\Gamma_2)\}) \\
&\quad \text{in } S \cup [\rho \cdot S(\Gamma_1')/\rho_1, \rho \cdot S(\Gamma_2')/\rho_2]
\end{aligned}$$

$$\begin{aligned}
\text{CUNIFY}(\Gamma_1, \Gamma_2) = \; & \text{if } dom(\Gamma_1) \neq dom(\Gamma_2) \text{ then } \mathit{failure} \\
& \text{else UNIFY}(\{(\Gamma_1(x), \Gamma_2(x)) \mid x \in dom(\Gamma_1) \cap dom(\Gamma_2)\})
\end{aligned}$$

$$\begin{aligned}
\text{CUNIFY}(\rho \cdot \Gamma_1, \Gamma_2) = \; & \text{if } dom(\Gamma_1) \not\subseteq dom(\Gamma_2) \text{ then } \mathit{failure} \\
& \text{else let } S = \text{UNIFY}(\{(\Gamma_1(x), \Gamma_2(x)) \mid x \in dom(\Gamma_1)\}) \\
& \quad\quad \text{in } S \cup [S(\Gamma_2|_{\overline{dom(\Gamma_1)}})/\rho]
\end{aligned}$$

$$\text{CUNIFY}(\Gamma_1, \rho \cdot \Gamma_2) = \text{CUNIFY}(\rho \cdot \Gamma_2, \Gamma_1)$$

Fig. 3. Context unification algorithm.

A substitution is then extended uniquely to any syntactic structures containing type variables and context variables. In what follows, we identify a substitution with its extension. The composition of substitutions S_1, S_2, denoted by $S_1 \circ S_2$, is defined as $S_1 \circ S_2(x) = S_1(S_2(x))$. We write $[v_1/x_1, \ldots, v_n/x_n]$ for a substitution that maps each x_i to v_i. We sometimes regard substitutions as sets of pairs and write $S_1 \cup S_2$ for the union of substitutions S_1 and S_2 provided that $dom(S_1) \cap dom(S_2) = \emptyset$.

Let UNIFY be a standard unification algorithm for types. Using UNIFY, we define a unification algorithm for contexts. Although context terms are similar to record types, we do not need any special machinery for record unification such as [17,12] due to restricted nature of our code language. We say that a set A of contexts is *well sorted* if whenever $\rho \cdot \Gamma_1$ and $\rho \cdot \Gamma_2$ appear in A, then $dom(\Gamma_1) = dom(\Gamma_2)$. We say that a substitution S is *admissible for* a set A of contexts if for any $\rho \cdot \Gamma \in A$, $dom(S(\rho)) \cap dom(\Gamma) = \emptyset$, and the set of context variables in the range of S is disjoint from the set of context variables in A. To infer a proof for our code language, we have only to consider well-sorted proof schemes and admissible substitutions for them.

Well-sortedness is preserved by admissible substitutions. The following property follows from the definitions.

Lemma 1. *If a set A of contexts is well sorted and S is an admissible substitution for A then $S(A)$ is also well sorted.*

Fig. 3 gives a unification algorithm for well-sorted pairs of contexts. The following property can be easily shown.

Lemma 2. *Let* (γ_1, γ_2) *be a well-sorted pair of contexts. If* CUNIFY$(\gamma_1, \gamma_2) = S$ *then* S *is admissible for* γ_1 *and* γ_2 *and* $S(\gamma_1) = S(\gamma_2)$. *Conversely, if there is an admissible substitution* S *for* γ_1 *and* γ_2 *such that* $S(\gamma_1) = S(\gamma_2)$ *then* CUNIFY$(\gamma_1, \gamma_2) = S'$ *such that* $S = S'' \circ S'$ *for some* S''.

Using CUNIFY and UNIFY, we define a proof reconstruction algorithm INFER. To present the algorithm, we first define some notations. In writing a proof tree, we only include, at each inference step, the instruction that is introduced (i.e. the first instruction of the sub-block). We write $\Delta(\Gamma \triangleright I : \tau)$ if Δ is a proof whose end sequent (i.e. the last statement in the proof) is $\Gamma \triangleright I : \tau$ (i.e. I is the instruction introduce by the last inference step.) If γ is a context containing x, $\gamma\{x : \tau\}$ is the context obtained from γ by replacing the value of x with τ. We also write $\bar{x}$ and $\overline{x : \tau}$ for a sequence of variables and a sequence of typed variables.

The proof reconstruction algorithm we shall define computes a proof scheme for a basic block together with a set of constraints imposed by branching instructions. An *entry constraint* is a sentence of the form $l \ll \gamma \triangleright \tau$. An *inclusion constraint* is a sentence of the form $\gamma \triangleright \tau \ll \gamma' \triangleright \tau'$. We say that a substitution S satisfies a set $\mathscr{S}$ of inclusion constraints if for each $(\gamma \triangleright \tau \ll \gamma' \triangleright \tau') \in \mathscr{S}$, $S(\gamma) \subseteq S(\gamma')$ and $S(\tau) = S(\tau')$. Let $\mathscr{L}$ be a label environment. We say that C is a set of entry constraints for $\mathscr{L}$ if all the labels mentioned in C are defined in $\mathscr{L}$. If C is a set of entry constraints for $\mathscr{L}$ then $\mathscr{L}(C)$ is the set of inclusion constraints obtained from C by replacing each l appearing in C with $\mathscr{L}(l)$. We say that $(\mathscr{L}, S)$ is a *solution* of C if $\mathscr{L}$ does not contain context variable, C is a set of entry constraints for $\mathscr{L}$ and S satisfies $\mathscr{L}(C)$.

The algorithm INFER is given in Fig. 4, which uses sub-algorithms SOLVE and INFBLK (given in Fig. 5.) The main algorithm takes a labeled set of basic blocks $\{l_1 : B_1, \ldots, l_n : B_n\}$ and returns a labeled set of proofs. It first uses INFBLK to infer for each B_i its proof scheme Δ_i (i.e. a proof containing context variables) together with a set C_i 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 of 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 $\mathscr{L}(C_1 \cup \cdots \cup C_n)$ of inclusion constraints, and solves it by fixed point computation, where each iteration step picks one sentence $\rho' \cdot \Gamma' \triangleright t' \ll \rho \cdot \Gamma \triangleright t$ 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, INFER instantiates all the context variables with the empty set to obtain a ground proof.

We establish the soundness of INFER by showing the correctness of each sub-algorithm used.

The following property is necessary to show that the set of entry constraints has a solution.

$\text{INFER}(\{l_1 : B_1, \ldots, l_n : B_n\}) =$
  let $(C_i, \Delta_i(\gamma_i \rhd \tau_i)) = \text{INFBLK}(B_i) \quad (1 \leqslant i \leqslant n)$
    $\mathscr{L} = \{l_1 : \gamma_1 \rhd \tau_1, \ldots, l_n : \gamma_n \rhd \tau_n\}$
    $S = \text{SOLVE}(\mathscr{L}(C_1 \cup \cdots \cup C_n))$
    $\mathscr{P} = S(\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\})$
    $\{\rho_1, \ldots, \rho_k\} = \mathit{FreeContextVars}(\mathscr{P})$
  in $[\emptyset/\rho_1, \ldots, \emptyset/\rho_k](\mathscr{P})$

$\text{SOLVE}(\mathscr{S}) =$
  if for each $(\gamma_1 \rhd \tau_1, \gamma_2 \rhd \tau_2) \in \mathscr{S}$, $\gamma_1 \subseteq \gamma_2$ and $\tau_1 = \tau_2$
  then $\emptyset$
  else if there is some $(\rho_1 \cdot \Gamma_1 \rhd \tau_1 \ll \rho_2 \cdot \Gamma_2 \rhd \tau_2) \in \mathscr{S}$ s.t.
      $\Gamma_2(x) = \texttt{nil}$ and $\Gamma_1(x) \neq \texttt{nil}$ for some x
  then *failure*
  else
    let $S_1 = \text{UNIFY}(\{(\Gamma_1(x), \Gamma_2(x)) \mid x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_1)\} \cup \{(\tau_1, \tau_2)\})$
      $\Gamma_3 = \{x : \Gamma_1(x) \mid x \in (\mathit{dom}(\Gamma_1) \backslash \mathit{dom}(\Gamma_2)), \Gamma_1(x) \neq \texttt{nil}\}$
      $S_2 = [\rho_3 \cdot S_1(\Gamma_3)/\rho_2] \cup S_1 \quad (\rho_3 \text{ fresh})$
      $S_3 = \text{SOLVE}(S_2(\mathscr{S}))$
    in $S_3 \circ S_2$

Fig. 4. The proof reconstruction algorithm for programs.

Lemma 3. *If* $\text{INFBLK}(B) = (C, \Delta)$ *then both* C *and* Δ *are well sorted, and* γ *appearing in* C *does not contain entries of the form* $x : \texttt{nil}$.

Proof. The first property is shown by simple induction using Lemma 1 and the fact that any context variables introduce by CUNIFY and INFBLK are fresh. For the second property, we note that any conditional branch instruction in B is followed by `goto` statement. The property can then be show by induction on B. □

We define the *type erasure* of a proof Δ to be the raw code obtained from Δ by erasing types and `discard` instructions. The following is the main lemma for the soundness of the proof inference algorithm.

Lemma 4. *Let* B *be a basic block. If* $\text{INFBLK}(B) = (C, \Delta)$ *then* $\mathit{erasure}(\Delta) = B$, *and for any solution* $(\mathscr{L}, S)$ *of* C, $\mathscr{L} \vdash_{SSC} S(\Delta)$.

Proof. The property of $\mathit{erasure}(\Delta) = B$ is shown by straightforward induction on the structure of B.

We show the second property by induction on B.

Case $B = \texttt{return}\ x$. By the typing rule, for any S, $\mathscr{L} \vdash_{SSC} S(\rho)\{x : S(t)\} \rhd \texttt{return}\ x : S(t)$.

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

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

$$\begin{array}{l}
\textsc{InfBlk}(\texttt{if } x \texttt{ then } l; B) = \\
\quad \text{let } (C_1, \Delta_0(\gamma_0 \triangleright I_0 : t_0)) = \textsc{InfBlk}(B) \\
\qquad S = \textsc{cunify}((\gamma_0, \rho_1 \cdot \{x : t_1\})) \quad (\rho_1, t_1 \text{ fresh}) \\
\quad \text{in } \left(S(C_1) \cup \{l \ll S(\gamma_0) \triangleright S(t_0)\}, \dfrac{S(\Delta_0)}{S(\gamma_0) \triangleright \texttt{if } x \texttt{ goto } l : S(t_0)} \right) \\
\textsc{InfBlk}(x \texttt{ = } v; B) = \\
\quad \text{let } (C_1, \Delta_0(\gamma_0 \triangleright I_0 : t_0)) = \textsc{InfBlk}(B) \\
\qquad S = \textsc{cunify}(\gamma_0, \rho_1 \cdot \overline{y : t_2}) \quad (\bar{y} = FV(v) \cup \{x\}, \text{and} \rho_1, \overline{t_2} \text{ fresh}) \\
\qquad \{y_1, \ldots, y_k\} = \{y' \mid (y' : \texttt{nil}) \in S(\gamma_0), y' \in \bar{y}\} \\
\qquad \Delta_1(\gamma_1 \triangleright \tau_1) = S(\Delta_0) \\
\qquad \Delta_{i+1}(\gamma_{i+1} \triangleright \tau_{i+1}) = \dfrac{\Delta_i}{\gamma_i\{y_i : t_i'\} \triangleright \texttt{discard } y_i : \tau_i} \quad (t_i' \text{ fresh}, 1 \leqslant i \leqslant k) \\
\quad \text{in if } x \in FV(v) \text{ then } \left(S(C_1), \dfrac{\Delta_{k+1}}{\gamma_{k+1}\{x : t_{k+1}\} \triangleright x\texttt{=}v : S(t_0)} \right) \quad (t_{k+1} \text{ fresh}) \\
\qquad \text{else } \left(S(C_1), \dfrac{\Delta_{k+1}}{\gamma_{k+1}\{x : \texttt{nil}\} \triangleright x\texttt{=}v : S(t_0)} \right)
\end{array}$$

Fig. 5. The proof reconstruction algorithm for blocks.

Case $B = \texttt{goto } l$. Let $(\mathscr{L}, S)$ be a solution of $\{l \ll \rho \triangleright t\}$. Then $\mathscr{L}(l) = \Gamma \triangleright t'$ such that $\Gamma \subseteq S(\rho)$ and $t' = S(t)$, and therefore, by the typing rule, $\mathscr{L} \vdash_{SSC} S(\rho) \triangleright \texttt{goto } l : S(t)$.

Case $B = \texttt{if } x \texttt{ goto } l; B$. Let $(\mathscr{L}, S_0)$ be a solution of $S(C_1) \cup \{l \ll S(\gamma_0) \triangleright S(t_0)\}$. Then $(\mathscr{L}, S_0 \circ S)$ is a solution of C_1. By the induction hypothesis, $\mathscr{L} \vdash_{SSC} S_0(S(\Delta_0))$. By Lemmas 1 and 2, $x \in dom(S(\gamma_0))$ and $S(\gamma_0)(x) \neq \texttt{nil}$. Then, since $(\mathscr{L}, S_0)$ is a solution of $\{l \ll S(\gamma_0) \triangleright S(t_0)\}$, $\mathscr{L}(l) = \Gamma \triangleright t$ such that $\Gamma \subseteq S_0(S(\gamma_0))$ and $t = S_0(S(t_0))$, and therefore by the typing rule,

$$\mathscr{L} \vdash_{SSC} \frac{S_0(S(\Delta_0))}{S_0(S(\gamma_0)) \triangleright \texttt{if } x \texttt{ goto } l : S_0(S(t_0))}.$$

Case $B = x \texttt{ = } v; B$. Let $(\mathscr{L}, S_0)$ be a solution of $S(C_1)$. Then $(\mathscr{L}, S_0 \circ S)$ is a solution of C_1. By the induction hypothesis, $\mathscr{L} \vdash_{SSC} S_0(S(\Delta_0))$. By the rule for `discard`, $\mathscr{L} \vdash_{SSC} S_0(\Delta_i)$ for each $1 \leqslant i \leqslant k + 1$. Moreover, for each $y \in FV(v)$, $y : t \in \gamma_{k+1}$ for some t. Then by the typing rules for assignment, if $x \in FV(v)$, then

$$\mathscr{L} \vdash_{SSC} \frac{S(\Delta_{k+1})}{S(\gamma_{k+1}\{x : t_{k+1}\}) \triangleright x \texttt{ = } v : S(t_0)}.$$

If $x \notin FV(v)$, then

$$\mathscr{L} \vdash_{SSC} \frac{S(\Delta_{k+1})}{S(\gamma_{k+1}\{x:\mathtt{nil}\}) \triangleright x \; \mathtt{=} \; v : S(t_0)}. \quad \square$$

Lemma 5. *If* Solve($\mathscr{S}$) *terminates with* S *and* $\{\rho_1, \ldots, \rho_k\}$ *is the set of free context variables in* $S(\mathscr{S})$ *then* $[\emptyset/\rho_1, \ldots, \emptyset/\rho_k] \circ S$ *satisfies* $\mathscr{S}$.

Proof. This is shown by induction on the length of computation (i.e. the number of recursive calls of Solve.) The basis is trivial. Suppose there is some $(\rho_1 \cdot \Gamma_1 \triangleright \tau_1 \ll \rho_2 \cdot \Gamma_2 \triangleright \tau_2) \in \mathscr{S}$ s.t. $\Gamma_1 \not\subseteq \Gamma_2$ or $\tau_1 \neq \tau_2$. By the property of unify and the construction of S_2, $S_2(\rho_1 \cdot \Gamma_1) \subseteq S_2(\rho_2 \cdot \Gamma_2)$. Therefore, if S_3 satisfies $S_2(\mathscr{S})$ then $S_2 \circ S_2$ satisfies $\mathscr{S}$. By the induction hypothesis, $[\emptyset/\rho_1, \ldots, \emptyset/\rho_k] \circ S_3$ satisfies $S_2(\mathscr{S})$. Hence $[\emptyset/\rho_1, \ldots, \emptyset/\rho_k] \circ S_3 \circ S_2$ satisfies $\mathscr{S}$. $\square$

Different from usual type inference algorithms, this proof reconstruction algorithm always succeeds for any given raw code and return a proof, which is a liveness-annotated code, as shown in the following.

Lemma 6. *For any program* P, Infer(P) *always succeeds.*

Proof. Since there is only one type constant, unify always succeeds. So does cunify for any well-sorted pairs. Thus InfBlk(B) always succeeds. From Lemmas 1 and 3, the call of Solve in Infer does not return *failure*. So we have only to show that Solve terminates on all the inputs. This is shown by observing the fact that the set of variables appearing in a given set of inclusion constraints is finite. $\square$

For a proof $\{l_1 : \Delta_1, \ldots, l_n : \Delta_n\}$ of a program, its type erasure is $\{l_1 : erase(\Delta_1), \ldots, l_n : erase(\Delta_n)\}$. Using these lemmas, we can show the following soundness theorem.

Theorem 7. *For any raw program* P, Infer(P) *succeeds with a proof* $\mathscr{P}$ *such that the erasure of* $\mathscr{P}$ *is* P *and* $\vdash_{SSC} \mathscr{P}$.

Proof. The first property is by Lemma 6. Let S, C_i, L be those mentioned in the algorithm, $S' = [\emptyset/\rho_1, \ldots, \emptyset/\rho_k] \circ S$, and $\mathscr{L}_0 = S'(\mathscr{L})$. We note that for two sets of inclusion constraints $\mathscr{S}_1, \mathscr{S}_2$, if $\mathscr{S}_1 \subseteq \mathscr{S}_2$ and $(\mathscr{L}_0, S')$ is a solution of $\mathscr{S}_2$ then $(\mathscr{L}_0, S')$ is also a solution of $\mathscr{S}_1$. By Lemma 5, S' satisfies $\mathscr{L}(C_i)$, i.e. $(\mathscr{L}_0, S')$ is a solution of C_i. The result then follows from Lemma 4. $\square$

As we shall see in the next section, however, the type inference algorithm does not compute "most general proof" or "minimal proof".

(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 it into basic blocks

$$
\begin{array}{l}
\{l_1\text{: } \mathtt{i\ =\ 1;\ s\ =\ 0;\ goto}\ l_2,\\
\ l_2\text{: } \mathtt{c\ =\ i\ >\ n;\ if\ c\ goto}\ l_4\mathtt{;\ goto}\ l_3,\\
\ l_3\text{: } \mathtt{s\ =\ s\ +\ i;\ i\ =\ i\ +\ 1;\ goto}\ l_2,\\
\ l_4\text{: } \mathtt{return\ s}\}
\end{array}
$$

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

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

$$
l_2\text{: } \cfrac{\cfrac{\rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\} \rhd \mathtt{goto}\ l_3 : t_4}{\rho_2\{\mathtt{c}:t_5,\mathtt{i}:t_6,\mathtt{n}:t_7\} \rhd \mathtt{if\ c\ goto}\ l_4 : t_4}}{\rho_2\{\mathtt{c}:\mathtt{nil},\mathtt{i}:t_6,\mathtt{n}:t_7\} \rhd \mathtt{c{=}i{>}n} : t_4}
$$

$$
l_4 : \quad \rho_4\{\mathtt{s}:t_{13}\} \rhd \mathtt{return\ s} : t_{13}
$$

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

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

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

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

$$
l_4 : \quad \{\mathtt{s}:t_1\} \rhd \mathtt{return\ s} : t_1
$$

Fig. 6. Example code and the reconstructed liveness proof.

Fig. 6 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.

We will use the sample source code (1) in Fig. 6 as our running example and will show examples of the proof transformation steps that follow (4) in Fig. 6. The examples above and those shown later are (reformatted) actual outputs of our prototype system.

4. Optimizing live ranges by inserting weakening rules

The proof system *SSC* implicitly includes weakening rules (`discard` instruction) in the initial sequents (rules for `return` and `goto`). Our proof reconstruction algorithm inserts weakening only when required by assignments; all the other weakening rules are included in `goto` and `return` instructions. As a consequence, reconstructed proofs are not optimal with respect to live ranges. The next step in our register allocation method is to transform the inferred proof so that all the weakening rules are explicit, and to optimize the places of these rules by proof normalization.

We define a new proof system, which we call $SSC(W)$, by refining *SSC* as follows.

- Empty entries of the form $x:\texttt{nil}$ are eliminated from each context since they are no longer needed after proof reconstruction. All the proof rules that mention `nil` are changed accordingly. For example, the rule for `discard` becomes

$$\frac{\Gamma \triangleright B \,:\, t_0}{\Gamma, x : t \triangleright \texttt{discard}\ x; B \,:\, t_0}$$

  as in the standard weakening rule in the Gentzen's sequent calculus.
- The rules for initial sequents (rules for `return` and `goto`) are changed to the following

$$\{x : t\} \triangleright \texttt{return}\ x \,:\, t \qquad \Gamma \triangleright \texttt{goto}\ l \,:\, \tau \quad (\text{if } \mathscr{L}(l) = \Gamma \triangleright \tau)$$

  so that they do not contain redundant assumptions.

In this proof system, only those variables that are used by the continuation of a basic block are live at the end of the block. All the other variables must be discarded explicitly before the branch instruction which terminates the basic block.

The resulting proof system $SSC(W)$ still has the freedom in terms of the places where `discard` instructions are inserted. Let I be an instruction of either of the forms: "$x\ \texttt{=}\ v$" or "$\texttt{if}\ x\ \texttt{goto}\ l$". We define the set $RefVars(I, \mathscr{L})$ of *variables referenced by I* under $\mathscr{L}$ as follows:

$$\begin{aligned} RefVars(x\ \texttt{=}\ v, \mathscr{L}) &= FV(v) \cup \{x\}, \\ RefVars(\texttt{if}\ x\ \texttt{goto}\ l, \mathscr{L}) &= dom(\Gamma) \cup \{x\}, \end{aligned}$$

where $\Gamma \triangleright \tau = \mathscr{L}(l)$.

The flexibility of places of `discard` instructions is characterized as commutative conversion of proofs. If $x \notin RefVars(I, \mathscr{L})$, then $SSC(W)$ admits the following

conversion:

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Gamma \rhd B : \tau_0\end{matrix}}{\Gamma, x:\tau \rhd \texttt{discard}\ x; B : \tau_0}}{\Gamma', x:\tau \rhd I; \texttt{discard}\ x; B : \tau_0} \quad \Longleftrightarrow \quad \dfrac{\dfrac{\begin{matrix}\vdots\\ \Gamma \rhd B : \tau_0\end{matrix}}{\Gamma' \rhd I; B : \tau_0}}{\Gamma', x:\tau \rhd \texttt{discard}\ x; I; B : \tau_0}$$

Since the order of consecutive `discard` instructions are irrelevant, we consider proofs module reordering of consecutive `discard` instructions. We define a reduction relation on proofs, $\mathscr{L} \vdash \Delta \longrightarrow \Delta'$, as the above conversion relation restricted to the left-to-right direction, and write $\mathscr{L} \vdash \Delta \stackrel{*}{\longrightarrow} \Delta'$ for its reflexive transitive closure.

The next step in our register allocation process is to optimize live ranges by the combination of the following:

(1) *Proof transformation from SSC to SSC(W)*: A reconstructed proof in *SSC* is transformed to one in *SSC*(W) by restricting the domain of the assumption set Γ at the `return` and `goto` to those actually live at the target code, and inserting `discard` instructions for the rest of variables immediately before these branching instructions.
(2) *Proof normalization in SSC(W)*: The explicitly inserted `discard` instructions at the above step are moved toward the root of the proof tree (i.e. toward the entry point of the code) by applying the reduction defined above so that variables are discarded as early as possible.

The optimization algorithm Weaken is given in Fig. 7, which performs the above two step simultaneously using sub-algorithms AddWeaken and WK.

The observant reader may have noticed that if Δ is a proof reconstructed by Infer and $\mathrm{WK}(\Delta(\Gamma \rhd _ : _)) = \Delta'(\Gamma' \rhd _ : _)$ then $\Gamma = \Gamma'$, and therefore the call of AddWeaken(Γ_i, Δ'_i) in Weaken is redundant. This step is there to make the algorithm general. As we see in the proof of correctness of the algorithm (Lemma 8 below) this generality is also needed.

If Δ is a proof in *SSC*, we write $\mathscr{W}(\Delta)$ for the proof in *SSC*(W) obtained from Δ by eliminating entries of the form x : `nil`, changing all the initial sequents so that they conform to the rules of *SSC*(W), and, for each initial sequent, inserting immediately after the initial sequent an inference step of `discard` x for each variable x that is eliminated from the context of the original sequent. We show the following.

Lemma 8. *For any proof* $\Delta(\Gamma \rhd _ : _)$ *in SSC under* $\mathscr{L}$, *if* $\mathrm{WK}(\Delta) = \Delta'$ *then* $\mathscr{L} \vdash \mathscr{W}(\Delta) \stackrel{*}{\longrightarrow}$ AddWeaken(Γ, Δ').

Proof. This is proved by induction on the structure of Δ. We distinguish cases in term of the last inference step in Δ.

$\textsc{Weaken}(\{l_1 : \Delta_1(\Gamma_1 \triangleright _ : _), \ldots, l_n : \Delta_n(\Gamma_n \triangleright _ : _)\}) =$
let $E = \{l_1 : dom(\Delta_1), \ldots, l_n : dom(\Delta_n)\}$
in $\{l_1 : \textsc{AddWeaken}(\Gamma_1, \mathrm{WK}(\Delta_1)), \ldots, l_n : \textsc{AddWeaken}(\Gamma_n, \mathrm{WK}(\Delta_n))\}$

$\textsc{AddWeaken}(\Gamma, \Delta_0(\Gamma_0 \triangleright _ : t)) =$
let $\{x_1 : t_1, \ldots, x_k : t_k\} = \Gamma|_{\overline{dom(\Gamma_0)}}$

$$\Delta_i(\Gamma_i \triangleright _ : t) = \frac{\Delta_{i-1}}{\Gamma_{i-1}\{x_i : t_i\} \triangleright \texttt{discard}\ x_i : t} \quad (1 \leqslant i \leqslant k)$$

in Δ_k

In the following definition, E is globally defined in Weaken above.

$\mathrm{WK}(\Gamma \triangleright \texttt{return}\ x : t) = \{x : \Gamma(x)\} \triangleright \texttt{return}\ x : t$

$\mathrm{WK}(\Gamma \triangleright \texttt{goto}\ l : t) = \Gamma|_{E(l)} \triangleright \texttt{goto}\ l : t$

$$\mathrm{WK}\left(\frac{\Delta}{\Gamma \triangleright \texttt{if}\ x\ \texttt{goto}\ l : t}\right) =$$

let $\Delta'_0(\Gamma'_0 \triangleright _ : _) = \mathrm{WK}(\Delta)$
$\{x_1, \ldots, x_n\} = (E(l) \cup \{x\}) \setminus dom(\Gamma'_0)$

$$\Delta'_i(\Gamma'_i \triangleright _ : _) = \frac{\Delta'_{i-1}}{\Gamma'_{i-1}\{x_i : \Gamma(x_i)\} \triangleright \texttt{discard}\ x_i : t} \quad (1 \leqslant i \leqslant n)$$

in $\dfrac{\Delta'_n}{\Gamma'_n \triangleright \texttt{if}\ x\ \texttt{goto}\ l : t}$

$$\mathrm{WK}\left(\frac{\Delta(\Gamma_1 \triangleright _ : _)}{\Gamma_2 \triangleright x\ \texttt{=}\ v : t}\right) =$$

let $\Delta'_0(\Gamma'_0 \triangleright I_0 : t) = \mathrm{WK}(\Delta)$
$\{x_1, \ldots, x_n\} = FV(v) \setminus dom(\Gamma'_0)$

$$\Delta'_i(\Gamma'_i \triangleright _ : _) = \frac{\Delta'_{i-1}}{\Gamma'_{i-1}\{x_i : \Gamma_1(x_i)\} \triangleright \texttt{discard}\ x_i : t} \quad (1 \leqslant i \leqslant n)$$

in if $x \in FV(v)$ then $\dfrac{\Delta'_n}{\Gamma'_n\{x : \Gamma_2(x)\} \triangleright x\ \texttt{=}\ v : t}$

else $\dfrac{\Delta'_n}{\Gamma'_n|_{\bar{x}} \triangleright x\ \texttt{=}\ v : t}$

$$\mathrm{WK}\left(\frac{\Delta}{\Gamma \triangleright \texttt{discard}\ x : \tau}\right) = \mathrm{WK}(\Delta)$$

Fig. 7. Weakening (discard pseudo-instruction) insertion algorithm for blocks.

Case $\Delta = \Gamma \triangleright \texttt{return}\ x : t$: $\mathrm{WK}(\Delta) = \{x : t\} \triangleright \texttt{return}\ x : t$. Then by definition $\mathscr{W}(\Delta) = \textsc{AddWeaken}(\Gamma, \{x : t\} \triangleright \texttt{return}\ x : t)$. The case for `goto` l is similar.

Case

$$\Delta = \frac{\Delta_0(\Gamma_0 \triangleright _ : _)}{\Gamma \triangleright \texttt{if}\ x\ \texttt{goto}\ l : t}.$$

Suppose $\Delta_0'(\Gamma_0' \rhd _ : _) = \mathrm{WK}(\Delta_0)$. Let $\{x_1, \ldots, x_n\} = (E(l) \cup \{x\}) \backslash dom(\Gamma_0')$, and $\{y_1, \ldots, y_k\} = dom(\Gamma) \backslash (dom(\Gamma_0') \cup \{x_1, \ldots, x_n\}$. By the induction hypothesis,

$$\mathscr{L} \vdash \mathscr{W}(\Delta_0) \xrightarrow{*} \textsc{AddWeaken}(\Gamma_0, \Delta_0') = \begin{array}{c} \Delta_0' \\ \hline \Gamma_{x_1} \rhd \texttt{discard}\ x_1 : t \\ \hline \vdots \\ \hline \Gamma_{x_n} \rhd \texttt{discard}\ x_n : t \\ \hline \Gamma_{y_1} \rhd \texttt{discard}\ y_1 : t \\ \hline \vdots \\ \hline \Gamma_{y_k} \rhd \texttt{discard}\ y_k : t \end{array}$$

Since $\Gamma_{y_k} = \Gamma_0 = \Gamma$, we have

$$\mathscr{L} \vdash \mathscr{W}\left(\frac{\Delta_0}{\Gamma \rhd \texttt{if}\ x\ \texttt{goto}\ l : t}\right) \xrightarrow{*} \begin{array}{c} \Delta_0' \\ \hline \Gamma_{x_1} \rhd \texttt{discard}\ x_1 : t \\ \hline \vdots \\ \hline \Gamma_{x_n} \rhd \texttt{discard}\ x_n : t \\ \hline \Gamma_{y_1} \rhd \texttt{discard}\ y_1 : t \\ \hline \vdots \\ \hline \Gamma_{y_k} \rhd \texttt{discard}\ y_k : t \\ \hline \Gamma \rhd \texttt{if}\ x\ \texttt{goto}\ l : t \end{array}$$

Since $y_i \notin RefVars(\texttt{if}\ x\ \texttt{goto}\ l, \mathscr{L})$, we have

$$\mathscr{L} \vdash \begin{array}{c} \Delta_0' \\ \hline \Gamma_{x_1} \rhd \texttt{discard}\ x_1 : t \\ \hline \vdots \\ \hline \Gamma_{x_n} \rhd \texttt{discard}\ x_n : t \\ \hline \Gamma_{y_1} \rhd \texttt{discard}\ y_1 : t \\ \hline \vdots \\ \hline \Gamma_{y_k} \rhd \texttt{discard}\ y_k : t \\ \hline \Gamma \rhd \texttt{if}\ x\ \texttt{goto}\ l : t \end{array} \xrightarrow{*} \begin{array}{c} \Delta_0' \\ \hline \Gamma_{x_1} \rhd \texttt{discard}\ x_1 : t \\ \hline \vdots \\ \hline \Gamma_{x_n} \rhd \texttt{discard}\ x_n : t \\ \hline \Gamma_{x_n} \rhd \texttt{if}\ x\ \texttt{goto}\ l : t \\ \hline \Gamma_{y_1} \rhd \texttt{discard}\ y_1 : t \\ \hline \vdots \\ \hline \Gamma_{y_k} \rhd \texttt{discard}\ y_k : t \end{array} = \Delta'$$

By the definition of the algorithm, $\Delta' = \textsc{AddWeaken}(\Gamma, \Delta)$.

The case for

$$\frac{\Delta_0(\Gamma_0 \rhd _ : _)}{\Gamma \rhd x\ \texttt{=}\ v : t}$$

is shown similarly to the above.

The case for

$$\frac{\Delta_0(\Gamma_0 \rhd _ : _)}{\Gamma \rhd \texttt{discard}\ x : t}$$

is trivial. □

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

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

$$l_4:\ \{\texttt{s}:t_1\}\rhd \texttt{return s}\ :\ t_1$$

Fig. 8. The result of weakening insertion optimization for the example in Fig. 6.

Fig. 8 shows the optimized proof of our running example in Fig. 6.

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.

5. Limiting the length of active contexts

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 is ad hoc in conventional frameworks. Our framework provides a systematic approach 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 variables needed by each instruction. In our case, instructions have at most 2 operands and therefore $k \geqslant 2$. Each logical rule (instruction) can only

access assumptions in Π. To assess Σ, we introduce instructions `load` and `store` to move assumptions between Σ and Π.

For a register context Π to represent the exact set of live variables at each instruction step, we eliminate explicit `discard` x instructions and implicitly combine them with other instruction. This is necessary since x is not live at `discard` x;B. For example, a code of the form

$$x \ \texttt{=}\ y \ \texttt{+}\ z;\texttt{discard}\ z;B$$

is converted to

$$x \ \texttt{=}\ y \ \texttt{+}\ z;B$$

which has the following derivation:

$$\cfrac{\cfrac{\vdots}{\Sigma \mid \Pi, x : t_1, y : t_2, \rhd_k B : t}}{\Sigma \mid \Pi, y : t_2, z : t_3, \rhd_k x \ \texttt{=}\ y \ \texttt{+}\ z;B : t}$$

In this derivation, the inference step for x `=` y `+` z implicitly discards z.

We call the new proof system $SSC(WE, k)$. As before, the proof rules are relative to a given label environment $\mathscr{L}$. In $SSC(WE, k)$, a label environment $\mathscr{L}$ assigns each label l, typing of the form $\Sigma_l \mid \Pi_l \rhd_k t$ such that $|\Pi_l| \leqslant k$. We write $\Sigma \mid \Pi \rhd_k \tau \ll \Sigma' \mid \Pi' \rhd_k \tau$ if $\Sigma \subseteq \Sigma'$ and $\Pi \subseteq \Pi'$. Fig. 9 gives the set of proof rules. We write $\mathscr{L} \vdash_{SSC(WE,k)} \Delta$ if Δ is a proof in $SSC(WE, k)$ under $\mathscr{L}$. The definition of a proof of a program is the same as before.

In the rules for assignments, x can be equal to y or z. In that case $x : t_1$ does not appear in the upper sequent. The rules for conditional jump and assignment potentially contain `discard`, which is expressed by the associated inclusion condition. For example, $\Pi' \subseteq x : t_1, y : t_2, z : t_3$ in the rule for x `=` y `+` z indicates that some of x, y, z can be discarded at this step. Note that x can also be discarded immediately after its definition. This happens if x is assigned but is never used. We shall comment on this issue when we describe machine code emission in Section 6.

In a proof-theoretical perspective, the previous proof system $SSC(W)$ implicitly admits unrestricted exchange so that any assumptions in Γ are freely available, while the new proof system $SSC(WE, k)$ 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 9. *There is an algorithm* Exchange *such that, for any provable program P in $SSC(W)$,* Exchange(k,P) *is a program in $SSC(WE, k)$, 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

$$\Sigma \mid \Pi, x:t \rhd_k \texttt{return}\ x : t \ (\text{if } |\Pi \cup \{x:t\}| \leqslant k)$$

$$\Sigma \mid \Pi \rhd_k \texttt{goto}\ l : t_0 \ (\text{if } \mathscr{L}(l) = \Sigma \mid \Pi \rhd_k t_0)$$

$$\frac{\Sigma \mid \Pi, \Pi' \rhd_k I : t_0}{\Sigma \mid \Pi, x:t \rhd_k \texttt{if}\ x\ \texttt{goto}\ l : t_0}$$
$$(\text{if } \mathscr{L}(l) \ll \Sigma \mid \Pi, x:t \rhd_k t_0;\ \Pi' \subseteq \{x:t\};\ |\Pi, x:t| \leqslant k)$$

$$\frac{\Sigma \mid \Pi, \Pi' \rhd_k I : t_0}{\Sigma \mid \Pi \rhd_k x\ \texttt{=}\ c : t_0} \ (\text{if } \Pi' \subseteq \{x:t\})$$

$$\frac{\Sigma \mid \Pi, \Pi' \rhd_k I : t_0}{\Sigma \mid \Pi, y:t_2 \rhd_k y\ \texttt{=}\ x : t_0} \ (\Pi' \subseteq \{x:t_1, y:t_2\};\ |\Pi, y:t_2| \leqslant k)$$

$$\frac{\Sigma \mid \Pi, \Pi' \rhd_k I : t_0}{\Sigma \mid \Pi, y:t_2, z:t_3 \rhd_k x\ \texttt{=}\ y\ \texttt{+}\ z : t_0}$$
$$(\Pi' \subseteq \{x:t_1, y:t_2, z:t_3\};\quad |\Pi, y:t_2, z:t_3| \leqslant k\ )$$

$$\frac{\Sigma \mid \Pi, x:t_1 \rhd_k I : t}{\Sigma, x:t_1 \mid \Pi \rhd_k \texttt{load}\ x : t_0}$$

$$\frac{\Sigma, x:t_1 \mid \Pi \rhd_k I : t_0}{\Sigma \mid \Pi, x:t_1 \rhd_k \texttt{store}\ x : t_0} \ (\text{if } |\Pi \cup \{x:t_1\}| \leqslant k)$$

A label environment $\mathscr{L}$ for this proof system must satisfy the following property: for all $l \in dom(\mathscr{L})$, $\mathscr{L}(l)$ is of the form $\Sigma_l \mid \Pi_l \rhd_k t$ such that $|\Pi_l| \leqslant k$.

Fig. 9. A proof system with explicit exchange: $SSC(WE, k)$.

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 proof-theoretical framework is only to represent various strategies, and it does not itself offer any efficient strategy.

In our prototype system, we adopt 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. We write UseOrder(V, Δ) for the reversal of the ordered list of variables V obtained by traversing Δ. Figs. 10 and 11 give an example of the exchange insertion algorithm based on this simple strategy. In practice, we need more sophisticated heuristics, which is outside of the scope of this paper.

$\text{EXCHANGE}(k, \{l_1 : \Delta_1(\Gamma_1 \triangleright I_1 : t_1), \ldots, l_n : \Delta_n(\Gamma_n \triangleright I_n : t_n)\}) =$
let k_i be $k-1$ if l_i is a target of some conditional branch otherwise k
V_i be the first (at most) k_i elements in $\text{USEORDER}(FV(B_i), \Delta_i)$
Π_i be the restriction of Γ_i on V_i
Σ_i be $\Gamma_i \setminus \Pi_i$
$E = \{l_1 : (\Sigma_1, \Pi_1), \ldots, l_n : (\Sigma_n, \Pi_n)\}$
in $\{l_1 : \text{EX}(\Sigma_1, \Pi_1, \Delta_1), \ldots, l_n : \text{EX}(\Sigma_n, \Pi_n, \Delta_n)\}$

E is made globally available in the definition of EX function.

Fig. 10. Exchange insertion algorithm for program.

Fig. 12 shows the result of exchange insertion for the optimized proof shown in Fig. 7.

6. Assigning register numbers

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), so that Π is of the form $\{x_1 : t_1[p_1], \ldots, x_n : t_n[p_n]\}$. A label environment $\mathscr{L}$ is also refined to be a function that maps labels to typings of the form $\Sigma \mid \Pi \triangleright_k t$ where Π is of the above form. The set of instructions in this final proof system is as follows:

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

`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 that changes the registers allocated to x, which corresponds to register–register copy instruction.

Note that we assign a register number not to a variable but to each liveness type variable t. Therefore, variables $x, y, \ldots$ in this code language do not contain register numbers. They appear in a proof as additional type attributes.

Fig. 13 gives a proof system with register number attributes, which we call $SSC(WEA, k)$. As in $SSC(WE, k)$, in the rules for assignments, x can be the same as y or z. In that case $x : t_1[p_1]$ does not appear in the upper sequent. We write $\mathscr{L} \vdash_{SSC(WEA,k)} \Delta$ if Δ is a proof of this proof system under $\mathscr{L}$.

Register number assignment is done by transforming a proof in $SSC(WE, k)$ to $SSC(WEA, k)$. Since the length of each register context Π in a proof is limited to k, it is obvious that there is a proof transformation. Within a block, it 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. An extra work is needed to adjust register assignment before a branching instruction

In the following definition, E is defined globally by the main algorithm.

$$\mathrm{EX}(\Sigma,\Pi,\Gamma \triangleright \texttt{return}\ x : t) = \text{if } x \in dom(\Pi) \text{ then } \emptyset \mid \{x : \Pi(x)\} \triangleright_k \texttt{return}\ x : t$$
$$\text{else } \frac{\emptyset \mid \{x : \Pi(x)\} \triangleright_k \texttt{return}\ x : t}{\{x : \Sigma(x)\} \mid \emptyset \triangleright_k \texttt{load}\ x : t}$$

$$\begin{aligned}
&\mathrm{EX}(\Sigma,\Pi,\Gamma \triangleright \texttt{goto}\ l : t) = \\
&\quad \text{let } (\Sigma_0,\Pi_0) = E(l) \\
&\qquad \{x_1,\ldots,x_m\} = dom(\Pi_0) \setminus dom(\Pi) \\
&\qquad \{y_1,\ldots,y_n\} = dom(\Sigma_0) \setminus dom(\Sigma) \\
&\qquad \Delta_0 = \Sigma_0 \mid \Pi_0 \triangleright_k \texttt{goto}\ l : t \\
&\qquad \Delta_i(\Sigma_i \mid \Pi_i \triangleright_k _ : _) = \\
&\qquad\quad \frac{\Delta_{i-1}}{\Sigma_{i-1}\{x_i : \Pi_{i-1}(x_i)\} \mid \Pi_{i-1}|_{\overline{\{x_i\}}} \triangleright_k \texttt{load}\ x_i : t} \quad (1 \leqslant i \leqslant m) \\
&\qquad \Delta_{m+j}(\Sigma_{m+j} \mid \Pi_{m+j} \triangleright_k _ : _) = \\
&\qquad\quad \frac{\Delta_{m+j-1}}{\Sigma_{m+j-1}|_{\overline{\{y_j\}}} \mid \Pi_{m+j-1}\{y_j : \Pi_{m+j-1}(y_j)\} \triangleright_k \texttt{store}\ y_j : t} \quad (1 \leqslant j \leqslant n) \\
&\quad \text{in } \Delta_{m+n}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{EX}\left(\Sigma,\Pi,\frac{\Delta}{\Gamma \triangleright x\ \texttt{=}\ v : t}\right) = \\
&\quad \text{let } \{x_1,\ldots,x_m\} = FV(v) \setminus dom(\Pi) \\
&\qquad n = m + (\text{length of } \Pi) - k \\
&\qquad y_1,\ldots,y_n \text{ be the last } n \text{ variables in } \textsc{UseOrder}(dom(\Pi) \setminus \{x_1,\ldots,x_m\},\Delta) \\
&\qquad \Sigma_0 = \Sigma|_{\overline{\{x_1,\ldots,x_m\}}}\{y_1 : \Pi(y_1),\ldots,y_n : \Pi(y_n)\} \\
&\qquad \Pi_0 = \Pi|_{\overline{\{y_1,\ldots,y_n\}}}\{x_1 : \Sigma(x_1),\ldots,x_m : \Sigma(x_m)\} \\
&\qquad \Sigma' = \text{if } x \in dom(\Pi) \text{ then } \Sigma_0 \text{ else } \Sigma_0|_{\overline{\{x\}}} \\
&\qquad \Pi' = \text{if } x \in dom(\Pi) \text{ then } \Pi_0 \text{ else } \Pi_0\{x : \Sigma(x)\} \\
&\qquad \Delta_0 = \frac{\mathrm{EX}(\Sigma',\Pi',\Delta)}{\Sigma_0 \mid \Pi_0 \triangleright_k x\ \texttt{=}\ v : t} \\
&\qquad \Delta_i(\Sigma_i \mid \Pi_i \triangleright_k _ : _) = \\
&\qquad\quad \frac{\Delta_{i-1}}{\Sigma_{i-1}\{x_i : \Pi_{i-1}(x_i)\} \mid \Pi_{i-1}|_{\overline{\{x_i\}}} \triangleright_k \texttt{load}\ x_i : t} \quad (1 \leqslant i \leqslant m) \\
&\qquad \Delta_{m+j}(\Sigma_{m+j} \mid \Pi_{m+j} \triangleright_k _ : _) = \\
&\qquad\quad \frac{\Delta_{m+j-1}}{\Sigma_{m+j-1}|_{\overline{\{y_j\}}} \mid \Pi_{m+j-1}\{y_j : \Pi_{m+j-1}(y_j)\} \triangleright_k \texttt{store}\ y_j : t} \quad (1 \leqslant j \leqslant n) \\
&\quad \text{in } \Delta_{m+n}
\end{aligned}$$

$$\mathrm{EX}\left(\Sigma,\Pi,\frac{\Delta}{\Gamma \triangleright \texttt{discard}\ x : t}\right) = \mathrm{EX}(\Sigma,\Pi|_{\bar{x}},\Delta)$$

The case for `if` x `goto` l is defined similarly to that for `goto` l.

Fig. 11. Exchange insertion algorithm for blocks.

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

l_1:

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

l_3:

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

l_2:

$$\dfrac{\emptyset \mid \{\mathtt{s}:t_1\} \rhd_3 \mathtt{return\ s} : t_1}{\{\mathtt{s}:t_1\} \mid \emptyset \rhd_3 \mathtt{load\ s} : t_1}$$

l_4:

Fig. 12. The result of exchange insertion for the running example.

$$\Sigma \mid \Pi, x:t[p] \rhd_k \mathtt{return}\ x : t \ (\text{if } |\Pi \cup \{x:t[p]\} \leqslant k|)$$

$$\Sigma \mid \Pi \rhd_k \mathtt{goto}\ l : t_0 \ (\text{if } \mathscr{L}(l) = \Sigma \mid \Pi \rhd_k t_0)$$

$$\frac{\Sigma \mid \Pi' \rhd_k I : t_0}{\Sigma \mid \Pi, x:t[p] \rhd_k \mathtt{if}\ x\ \mathtt{goto}\ l : t_0} \ (\text{if } \mathscr{L}(l) \ll \Sigma \mid \Pi, x:t[p] \rhd_k t;\quad \Pi' \subseteq \Pi, x:t[p])$$

$$\frac{\Sigma \mid \Pi, x:t[p] \rhd_k I : t_0}{\Sigma, x:t \mid \Pi \rhd_k \mathtt{load}\ (p,x) : t_0}$$

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

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

The other rules are obtained from the corresponding rules of $SSC(WE,k)$ by adding to each t distinct register number attribute p.

Fig. 13. A proof system with register number annotations: $SSC(WEA,k)$.

so that the assignment at the branching instruction agrees with that of the target block. If the target block is not yet processed, then the algorithm 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 the algorithm needs to permutate some registers by inserting `move` instructions using one temporary register. If there is no free register, it has to save one and then load after the permutation.

Minimizing register–register moves at branching instructions is a difficult problem, for which our theoretical framework does not directly offer any solution. 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. 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 $SSC(WEA, k)$. We consider the following target machine code:

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

`r`i is the register identified by number i. `store (r`i,x`)` stores the register `r`i to the memory location named x. `load (r`i,x`)` loads the register `r`i with the content of the memory location x. Since in a proof of $SSC(WEA, k)$, each occurrence of variable in its register context Π is associate with a register number, it is a mechanical matter to extract the target machine code by traversing a proof and converting each variable with the register number. If a variable is an r-value (to be referenced) then it is taken from the lower sequent of the proof rule, and if it is an l-value (to be assigned) then it is taken from the upper sequent of the proof rule. For example, for the proof

$$\frac{\Delta(\Sigma \mid \Pi, x : t_1[1], y : t_2[2], \rhd_k I \ : \ t_0)}{\Sigma \mid \Pi, x : t_3[3], y : t_2[2] \rhd_k x \texttt{ = } x \texttt{ + } y \ : \ t_0}$$

we emit instruction "`r1 = r3 + r2`" and then continue to emit code for the proof Δ.

There is one minor subtle point in code emission. If a variable will never used after assignment, then it is not appear in the context of its upper sequent. In that case, the instruction is not used and we simply suppress emission of the code. This is a simple case of dead code elimination, which we get for free in our proof transformation.

Fig. 14 shows the proof in $SSC(WEA, k)$ and the machine code for our running example, which are generated by our prototype implementation.

7. Conclusions and discussion

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 a proof system. Register

The proof with register number annotation.

$$l_1: \quad \cfrac{\cfrac{\emptyset \mid \{\texttt{i} : t_2[r1], \texttt{n} : t_3[r0], \texttt{s} : t_1[r2]\} \rhd_3 \texttt{goto}\ l_2 : t_1}{\emptyset \mid \{\texttt{i} : t_2[r1], \texttt{n} : t_3[r0]\} \rhd_3 \texttt{s=0} : t_1}}{\emptyset \mid \{\texttt{n} : t_3[r0]\} \rhd_3 \texttt{i=1} : t_1}$$

$$l_2: \quad \cfrac{\cfrac{\cfrac{\{\texttt{s} : t_1\} \mid \{\texttt{i} : t_2[r1], \texttt{n} : t_3[r0]\} \rhd_3 \texttt{goto}\ l_3 : t_1}{\{\texttt{s} : t_1\} \mid \{\texttt{c} : t_4[r2], \texttt{i} : t_2[r1], \texttt{n} : t_3[r0]\} \rhd_3 \texttt{if c goto}\ l_4 : t_1}}{\{\texttt{s} : t_1\} \mid \{\texttt{i} : t_2[r1], \texttt{n} : t_3[r0]\} \rhd_3 \texttt{c=i>n} : t_1}}{\emptyset \mid \{\texttt{i} : t_2[r1], \texttt{n} : t_3[r0], \texttt{s} : t_1[r2]\} \rhd_3 \texttt{store s} : t_1}$$

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

$$l_4: \quad \cfrac{\emptyset \mid \{\texttt{s} : t_1[r0]\} \rhd_3 \texttt{return s} : t_1}{\{\texttt{s} : t_1\} \mid \emptyset \rhd_3 \texttt{load s} : t_1}$$

The extracted machine code:

```
l1 :   r1 = 1
       r2 = 0
       goto l2

l2:    store r2,s
       r2 = r1 > r0
       if r2 goto l4
       goto l3

l3:    load r2,s
       r2 = r2 + r1
       r1 = r1 + 1
       goto l2

l4:    load r0,s
       return r0
```

Fig. 14. Example of register number assignment and code emission.

allocation process is then regarded as a proof transformation from a proof system 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 a proof theoretical framework for register allocation; there remain a number of issues to be investigated, including 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 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. Since our method is a form of a type system, it can smoothly be integrated in a static type system of a code language such as the one developed in [7]. By regarding our liveness types as attributes of types of conventional notion, 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 and register contexts of each sequent. We also believe that our method can be combined with other static verification systems for low-level code such as a static type system for access control [8].

Expressiveness: Our formalism covers the entire process of register allocation, and as a formalism, it appears to be more powerful than existing ones. 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 Ref. [1], for which the conventional graph coloring based approach cannot find an optimal coloring. Fig. 15 shows one simple example.

Liveness analysis and SSA-style optimization: The main strength of our method is the representation of properties of programs by types. Since we can freely define types and typing relations independent of variable names, this structure enables us to represent various static information required for register allocation systematically and precisely. For example, as far as liveness analysis is concerned, our system already contains the effect of SSA (static single assignment) transformation [5] 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.

Optimization and heuristics: As we have commented early, our proof-theoretical framework is only to offer a high-level and systematic description of a register allocation algorithm as a series of proof transformation, and it does not itself provide any particular efficient proof transformation strategies. In order to develop an efficient and practical register allocation algorithm, one need to incorporate various optimization and heuristics studied in literature. With these efforts, we believe that the approach presented here can serve as a framework for systematic development of a practical register allocation system.

(1) The source program

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

(2) The optimized liveness proof:

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

(3) The proof without `discard`:

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

(4) The proof in $SSC(WE,k)$:

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

(5) The proof in $SSC(WEA,k)$:

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

(6) The machine code

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

Fig. 15. Example for a code whose interference graph forms a "diamond"

Acknowledgements

The author thanks Toshimasa 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, L. Torczon, Improvements to graph coloring register allocation, ACM Trans. Programming Languages Systems 16 (3) (1994) 428–455.

[2] G.J. Chaitin, Register allocation & spilling via graph coloring. in: Proc. ACM Symp. Compiler Construction, Boston, MA, 1982, pp. 98–105.

[3] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, Register allocation via coloring, Comput. Languages 6 (1) (1981) 47–57.
[4] H.B. Curry, R. Feys, Combinatory Logic, vol. 1, North-Holland, Amsterdam, 1968.
[5] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Programming Languages Systems 13 (4) (1991) 451–490.
[6] J.-Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1) (1987) 1–102.
[7] T. Higuchi, A. Ohori, Java bytecode as a typed term calculus, in: Proc. ACM Internat. Conf. Principles and Practice of Declarative Programming, Pittsburgh, PA, 2002, pp. 201–211.
[8] T. Higuchi, A. Ohori, A static type system for JVM access control, in: Proc. ACM Internat. Conf. Functional Programming, Uppsala, Sweden, 2003, pp. 227–237.
[9] W. Howard, The formulae-as-types notion of construction, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Academic Press, New York, 1980, pp. 476–490.
[10] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, in: Proc. ACM Symp. Principles of Programming Languages, San Diego, CA, 1998.
[11] G. Necula, P. Lee, Proof-carrying code, in: Proc. ACM Symp. Principles of Programming Languages, San Diego, CA, 1998, pp. 106–119.
[12] A. Ohori, A polymorphic record calculus and its compilation, ACM Trans. Programming Languages Systems 17 (6) (1995) 844–895 (A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi").
[13] A. Ohori, The logical abstract machine: a Curry–Howard isomorphism for machine code, in: Proc. Internat. Symp. Functional and Logic Programming, Tsukuba, Japan, 1999.
[14] H. Ono, Proof-theoretic methods in nonclassical logic, an introduction, in: M. Takahashi, M. Okada, M. Dezani-Ciancaglini (Eds.), Theories of Types and Proofs, MSJ Memoirs, vol. 2, Mathematical Society of Japan, Tokyo, 1998, pp. 207–254 (Chapter 6).
[15] H. Ono, Y. Komori, Logics without the contraction rule, J. Symbolic Logic 50 (1) (1985) 169–201.
[16] M. Poletto, V. Sarkar, Linear scan register allocation, ACM Trans. Programming Languages Systems 21 (5) (1999) 895–913.
[17] D. Remy, Typechecking records and variants in a natural extension of ML, in: Proc. ACM Symp. Principles of Programming Languages, Austin, TX, 1989, pp. 242–249.
[18] R. Stata, M. Abadi, A type system for Java bytecode subroutines, in: Proc. ACM Symp. Principles of Programming Languages, San Diego, CA, 1998, pp. 149–160.
[19] A.S. Troelstra, Lectures on Linear Logic, CSLI Lecture Notes, vol. 29, Center for the Study of Language and Information, Stanford University, 1992.