

# Denotation-based Compositional Compiler Verification

ZHANG CHENG, Shanghai Jiao Tong University, China

JIYANG WU, Shanghai Jiao Tong University, China

DI WANG, Peking University, China

QINXIANG CAO\*, Shanghai Jiao Tong University, China

A desired but challenging property of compiler verification is compositionality in the sense that the compilation correctness of a program can be deduced from that of its substructures ranging from statements, functions, and modules incrementally. Previously proposed approaches have devoted extensive effort to module-level compositionality based on small-step semantics and simulation theories. This paper proposes a novel compiler verification framework based on denotational semantics for better compositionality. Specifically, our denotational semantics is defined by semantic functions that map a syntactic component to a semantic domain composed of multiple behavioral *sets*, and compiler correctness is defined by the behavioral refinement between semantic domains of the source and the target programs. Therefore, when proving compiler correctness, we can extensively leverage the algebraic properties of sets. Another important contribution is that our formalization of denotational semantics captures the full meaning of a program and bridges the gap between those based on conventional powerdomains and what realistic compiler verification actually needs. We demonstrate our denotation-based framework viable and practical by applying it to the verification of the front-end of CompCert and showing that the compositionality from the compilation correctness of sub-statements to statements, from functions to modules, and from modules to the whole program (i.e., module-level compositionality) can be achieved similarly.

Additional Key Words and Phrases: Compiler Verification, Denotational Semantics, Compositionality

## 1 INTRODUCTION

For several decades, research on compiler verification has received a lot of attention, especially with the advent of the well-known realistic verified C Compiler—CompCert [Leroy 2009a]. Specifically, CompCert translates programs written in a large subset of C language into optimized assembly code, going through multiple intermediate languages. For each of them, program behavior is formulated by a labeled state transition system according to the small-step operational semantics. The compiler correctness is then achieved by showing a backward simulation<sup>1</sup> asserting that every execution step of the target program can be simulated by several execution steps of the source program in a behaviorally consistent way, as shown in Fig. 1a.

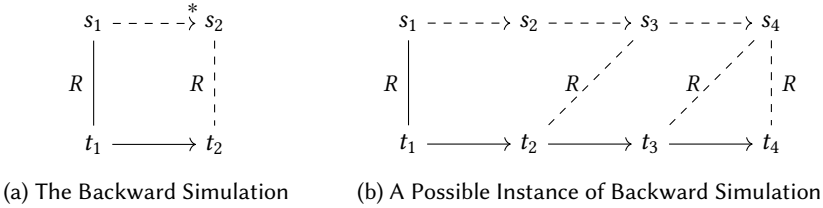


Fig. 1. Backward Simulation Diagrams Used in CompCert

In this paper, we propose a different framework for compiler verification—a denotation-based approach which focuses on the overall properties of programs and enjoys better proof compositionality. For example, considering type-safe and non-deterministic programs with the set of program

\*Corresponding author

<sup>1</sup>In fact, CompCert uses forward simulation to verify each compilation phase, and then flip that to backward simulation after vertical composition of it.

1a. while $e$ do $s$	2b. block {
1b. block {	block {
loop {	block {
block { if $e$ then $s$ else exit 1;	switch ( $e$ ) {
} //continue of $s$ branches here	N1: exit 0;
} //break of $s$ branches here	N2: exit 1;
2a. switch ( $e$ ) {	default: exit 2;
case N1: $s_1$ ;	} //exits shifted by 2
case N2: $s_2$ ;	}; $s_2$ //exits shifted by 1
default: $s$ ;	}; $s$ //exits unchanged
}	

Fig. 2. Clight program 1a and Csharpminor program 2a are respectively translated to Csharpminor program 1b and Cminor program 2b, where (exit  $n$ ) will prematurely terminate ( $n + 1$ ) layers of nested blocks. In program 2b, the exiting number in  $s_i$  is properly shifted according to the level of blocks it resides in.

states  $state$ , a textbook denotational semantics [Plotkin 1983, Chapter 8] for a program statement  $c$  can be defined by a subset of the binary relation on lifted program states (i.e.,  $\llbracket c \rrbracket \subseteq state \times state_{\perp}$ <sup>2</sup>), meaning that for any  $(\sigma_1, \sigma_2) \in \llbracket c \rrbracket$ , executing statement  $c$  from state  $\sigma_1$  may eventually terminate at state  $\sigma_2$  if  $\sigma_2 \neq \perp$ , or otherwise ( $\sigma_2 = \perp$ ) executing statement  $c$  from state  $\sigma_1$  may not terminate. Thus,

- the sequential statement's denotation can be defined by *composition* of binary relations, namely  $\llbracket c_1; c_2 \rrbracket \triangleq \llbracket c_1 \rrbracket \circ \llbracket c_2 \rrbracket$ ; and
- transformation correctness can be defined by the set inclusion, i.e., a transformation  $\mathcal{T}$  is sound iff.  $\forall c, \llbracket \mathcal{T}(c) \rrbracket \subseteq \llbracket c \rrbracket$ , which allows the transformation result  $\mathcal{T}(c)$  to have less possible behavior than the original program, but no extra behavior.

In this setting, if a transformation  $\mathcal{T}$  satisfies  $\mathcal{T}(c_1; c_2) = (\mathcal{T}(c_1); \mathcal{T}(c_2))$ , then its transformation correctness is compositional in terms of the composition relation, i.e.,

$$\text{if } \llbracket \mathcal{T}(c_1) \rrbracket \subseteq \llbracket c_1 \rrbracket \text{ and } \llbracket \mathcal{T}(c_2) \rrbracket \subseteq \llbracket c_2 \rrbracket, \text{ then } \llbracket \mathcal{T}(c_1) \rrbracket \circ \llbracket \mathcal{T}(c_2) \rrbracket \subseteq \llbracket c_1 \rrbracket \circ \llbracket c_2 \rrbracket \quad (1)$$

while it is not so straightforward to achieve this within the framework of operational semantics.

In fact, many compilation passes in CompCert are recursively defined program transformations, such as the transformation of **while**, **do .. while** and **for** loops into infinite loops with appropriate **block** and **exit** constructs, along with statements **break** and **continue** into early exits, and the transformation of multi-branch **switch** of Csharpminor into a Cminor **switch** wrapped by the statements associated with the various branches in a cascade of nested Cminor blocks, as Fig. 2 shows. When using the operational semantics-based approach to verify those recursively defined program transformations, one has to initially find a global simulation relation  $R$ , as shown in Fig. 1b, covering every intermediate correspondence, and such clutter of correspondence makes proofs unwieldy and brittle as well.

In contrast, our denotational approach simplifies this process and allows us to directly establish behavior refinement with a matching relation covering initial correspondence and ending correspondence only. Furthermore, the denotation of a statement is recursively defined over its syntax tree, and can be derived from its substructures with some unified semantic operators, e.g., the denotation of a sequential statement  $(c_1; c_2)$  is defined by the denotations of  $c_1$  and  $c_2$ , and the denotation of a loop statement is defined based on its loop body's. When verifying the correctness

<sup>2</sup>We denote  $state_{\perp} = state \cup \{\perp\}$ , where  $\perp$  represents non-termination.

of statement transformations, we can take induction over the syntax of source program statements and then produce proof obligations asserting that for each syntactic component, its corresponding semantic operators preserve transformation correctness. As illustrated by (1), this should be easy to prove with good algebraic properties of denotational semantics.

Although it's trivial to show transformation correctness with the textbook denotational semantics, we have to address the following key challenges when extending it to realistic compiler verification:

***How to tackle more program features and still keep good algebraic properties?*** Realistic denotational semantics has to take more program features into account, e.g., (i) handling traces of input-output events used to express a program's observable behaviors in CompCert, (ii) describing diverging, aborting and terminating behaviors simultaneously, (iii) manipulating a program's control flow, and (iv) building refinement relations between heterogeneous denotations of programs before and after compilation. Our denotational approach aims to exploit graceful algebraic properties of sets (e.g., the monotonicity and associativity of the composition relation) to facilitate compiler verification. If we extend the textbook denotational semantics into a realistic one, do similar algebraic properties still hold? If so, what is the theoretical reason behind it?

**Ans.** Despite various program features, we find that the composition and refinement relations for different program behaviors share common algebraic properties, which enable us to formalize them in a uniform way.

***How to distinguish different program behaviors precisely?*** Theoretically, defining a precise denotational semantics for programs with non-determinism and non-termination (i.e., unbounded non-determinism) turns out to be difficult. The early literature [Plotkin 1976; Smyth 1978; Winskel 1985] on the denotational semantics of non-deterministic programs has proposed three kinds of powerdomains known as the Hoare, Smyth, and Plotkin powerdomain. The word powerdomain means: the denotation of a program statement is a function from initial program states to the set of possible ending program states (i.e. the power set of program state set). These powerdomain constructions can work quite well in specific scenarios, but cannot be used directly for compiler verification since none of them can capture the full meaning of realistic languages. Specifically, Hoare powerdomain (for partial correctness) takes an angelic attitude toward non-termination, and it is usually used for defining Hoare triple validity w.r.t partial correctness. Therefore, it treats the same all programs that may not terminate. Smyth powerdomain (for total correctness) can be seen as the dual of Hoare powerdomain and takes a demonic attitude to non-termination. Plotkin powerdomain is a combination of them but limited to programs with bounded non-determinism [Back 1983]. In other words, none of the three powerdomains can describe the precise semantics of a language where a program may terminate, diverge, or abort when executing from a given initial state. Therefore, they are less expressive than existing operational semantics. To address the problem, later efforts have proposed two solutions: one is to divide the semantic domain of programs into different parts, and independently apply suitable fixed point theorems for each of them (see [Park 1979]); the other is to extend Plotkin's semantics but have to use transfinite induction for applying the Scott induction rule. Can we mechanically formalize the precise denotational semantics for realistic C programs by extending one of them?

**Ans.** Following Park's relational semantics [Park 1979], we choose to divide the semantic domain of programs into multiple fields and formalize this treatment in the Coq proof assistant with a record type, each field of which represents different program behaviors that are obtained via appropriate fixed-point theorems.

We aim to demonstrate the power of our denotation-based approach by **supporting module-level compositional compiler correctness**. Before this paper, there have been a series of efforts such as CompComp [Stewart et al. 2015], CompCertM [Song et al. 2020], CompCertO [Koenig and Shao 2021] and recent work by Zhang et al. [2023] trying to extend CompCert to support it. These approaches are all based on operational semantics (or, more precisely, interaction semantics), which makes it necessary to consider the correspondence between the whole stack involving every layer of function calls. We argue that this somehow increases the difficulty of compositionality proofs, when one expects to use different simulation relations to verify different compilation passes for each module and make all the proofs still compositional.

We present a very different method for supporting module-level compositionality, where a module’s semantic domain is parameterized by external callee’s behavior and the semantic linking of two modules (written as  $\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket$ ) is defined by taking fixed points on recursive domain equations. In this way, it becomes unnecessary for us to involve the whole stack in the compiler correctness proof but focus on the top layer of function calls. This approach also enables us to incorporate the idea of *calling convention* [Koenig and Shao 2021; Zhang et al. 2023] into our framework and simplify verification when using different simulation relations for each pass.

**THEOREM 1.1 (MODULE-LEVEL COMPOSITIONALITY).** *For any group of source modules  $S_1, \dots, S_n$  and target modules  $T_1, \dots, T_n$ , if  $\llbracket T_i \rrbracket \sqsubseteq \llbracket S_i \rrbracket$  for each  $i$ , then*

$$\llbracket T_1 + \dots + T_n \rrbracket \sqsubseteq \llbracket S_1 \rrbracket \oplus \dots \oplus \llbracket S_n \rrbracket$$

Our main theorem is then formulated as Thm. 1.1 where  $\sqsubseteq$  represents behavior refinement and  $+$  represents syntactic linking. This theorem is immediately deduced from two steps: (i) the semantic linking is monotonic in terms of behavior refinement, i.e., for any module  $M_1, M'_1, M_2$  and  $M'_2$ ,

$$\text{if } \llbracket M'_1 \rrbracket \sqsubseteq \llbracket M_1 \rrbracket \text{ and } \llbracket M'_2 \rrbracket \sqsubseteq \llbracket M_2 \rrbracket, \text{ then } \llbracket M'_1 \rrbracket \oplus \llbracket M'_2 \rrbracket \sqsubseteq \llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket;$$

and (ii) semantic linking is equivalent to syntactic linking, i.e., for any module  $M_1$  and  $M_2$ ,

$$\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket = \llbracket M_1 + M_2 \rrbracket.$$

We further find that the first step, the monotonicity of semantic linking (also known as horizontal compositionality), has the same form as the monotonicity of composition shown in (1), which indicates that we can prove them in the same way within the framework of denotational semantics.

To summarize, we develop a denotation-based framework for compositional compiler verification. To demonstrate its practicability, we define the denotational semantics for the front-end languages of CompCert, provide a structured technique to verify the compilation correctness from Clight to Cminor, and support module-level compositionality in a language-independent way. All our definitions and theorems in this paper are formalized and proved in the Coq proof assistant.

**Structure of the paper.** We organize the rest of this paper as follows.

- In §2 ~ §5, we develop the denotational semantics of programs starting from simple toy languages and generalizing to realistic languages. Especially, unified set operations are formalized in Coq for better proof reuse (§2); a novel semantic linking operator is proposed through Kleene fixed point and Knaster-Tarski fixed point, so that the connection between semantic linking and syntactic linking is reduced to concise lemmas about fixed points (§4).
- In §6 ~ §7, we propose a refinement algebra for unifying various behavior refinements between denotations, and reprove the compiler correctness of the CompCert front-end with our denotation-based approach—our proof supports module-level separate compilation.
- In §8~§10, we compare our work with existing research on compiler verification, discuss related work, and conclude this paper.

## 2 UNIFIED SEMANTIC OPERATORS

In this section, we begin with a toy language with non-deterministic and possible non-terminating behaviors, and show how we can use set operators and Park's approach [Park 1979] to define denotations of the toy language. Then we discuss how to extend this approach to realistic programming languages (e.g., Clight).

### 2.1 A Toy Language and its Denotation

Consider a toy language WHILE whose set of statements  $\text{Com}$ , ranged over by  $c$ , is parameterized on the set of Boolean expressions  $\text{Exp}$  ranged over by  $b$ , and generated by the following syntax:

$$c \triangleq \text{skip} \mid \text{atom} \mid \text{choice } c_1 \ c_2 \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c,$$

where *atom* represents the set of atomic statements that cannot be broken into other statements. Non-determinism can be introduced either by atomic statements like non-deterministic assignments or by **choice** statements that unpredictably choose  $c_1$  or  $c_2$  to execute.

Let *state* be the set of program states. We then define  $\text{CDenote}$  to represent the denotation of statements, and  $\text{BDenote}$  to represent the denotation of Boolean expressions as follows in Coq.

```
Record CDenote: Type := {
  nrm: state → state → Prop;      (* nrm ⊆ state × state *)
  dvg: state → Prop                (* dvg ⊆ state *)
}.
Record BDenote: Type := {
  tts: state → Prop;               (* tts ⊆ state *)
  ffs: state → Prop                (* ffs ⊆ state *)
}.
```

Here,  $\text{CDenote}$  is composed of two sets *nrm* and *dvg* which represent the set of terminating behaviors and diverging behaviors respectively. For  $\text{BDenote}$ , *tts* denotes the set of states satisfying expression  $b$  and *ffs* the set of states not satisfying  $b$ . Specifically,  $(\sigma_1, \sigma_2) \in \llbracket c \rrbracket.(\text{nrm})$  iff executing  $c$  from initial state  $\sigma_1$  could terminate on state  $\sigma_2$  and  $\sigma \in \llbracket c \rrbracket.(\text{dvg})$  iff executing  $c$  from  $\sigma_1$  could diverge;  $\sigma \in \llbracket b \rrbracket.(\text{tts})$  iff state  $\sigma$  satisfies  $b$  and  $\sigma \in \llbracket b \rrbracket.(\text{ffs})$  iff state  $\sigma$  does not satisfy  $b$ . Throughout the rest of the paper we will repeatedly overload the notation  $\llbracket \cdot \rrbracket$  to represent different denotations when there is no ambiguity.

### 2.2 Semantic Operators for WHILE

Naturally, the denotation of statements satisfies the following equations, where  $\mathbb{1}$  is the identity binary relation on *state*, i.e.,  $\{(\sigma, \sigma) \mid \sigma \in \text{state}\}$ , and  $\text{test}(X)$  defines an identity relation on the set  $X$ , i.e.,  $\{(x, x) \mid x \in X\}$ , meaning that the program state is not updated through them.

$$\begin{aligned} \llbracket \text{skip} \rrbracket.(\text{nrm}) &\triangleq \mathbb{1} & \llbracket \text{skip} \rrbracket.(\text{dvg}) &\triangleq \emptyset \\ \llbracket c_1; c_2 \rrbracket.(\text{nrm}) &\triangleq \llbracket c_1 \rrbracket.(\text{nrm}) \circ \llbracket c_2 \rrbracket.(\text{nrm}) \\ \llbracket \text{choice } c_1 \ c_2 \rrbracket.(\text{nrm}) &\triangleq \llbracket c_1 \rrbracket.(\text{nrm}) \cup \llbracket c_2 \rrbracket.(\text{nrm}) \\ \llbracket \text{choice } c_1 \ c_2 \rrbracket.(\text{dvg}) &\triangleq \llbracket c_1 \rrbracket.(\text{dvg}) \cup \llbracket c_2 \rrbracket.(\text{dvg}) \\ \llbracket \text{if } b \text{ then } c_1 \text{ else } c_2 \rrbracket.(\text{nrm}) &\triangleq \text{test}(\llbracket b \rrbracket.(\text{tts})) \circ \llbracket c_1 \rrbracket.(\text{nrm}) \cup \text{test}(\llbracket b \rrbracket.(\text{ffs})) \circ \llbracket c_2 \rrbracket.(\text{nrm}) \end{aligned}$$

The terminating case of **if** statements means that either the Boolean condition evaluates to true and then executing  $c_1$  terminates normally, or the condition evaluates to false and then executing  $c_2$  terminates normally. This style of defining if-branching is not new. Such formulations of test, sequential composition, and nondeterministic choices are widely used in extensions of Kleene algebras and in dynamic logics. Obviously, the diverging case of if statements should be defined

likewise as it has similar meanings to the terminating case. This description can be formalized with almost the same math formula if we overload the “ $\circ$ ” operator as follows.

$$\begin{aligned} R_1 \circ R_2 &\triangleq \{(\sigma_1, \sigma_3) \mid \exists \sigma_2, (\sigma_1, \sigma_2) \in R_1 \wedge (\sigma_2, \sigma_3) \in R_2\} && \text{original} \\ R \circ X &\triangleq \{\sigma_1 \mid \exists \sigma_2, (\sigma_1, \sigma_2) \in R \wedge \sigma_2 \in X\} && \text{overloaded} \end{aligned}$$

Then the diverging cases for sequential and if statements can be defined as:

$$\begin{aligned} \llbracket c_1; c_2 \rrbracket.(\text{dvg}) &\triangleq \llbracket c_1 \rrbracket.(\text{dvg}) \cup \llbracket c_1 \rrbracket.(\text{nrm}) \circ \llbracket c_2 \rrbracket.(\text{dvg}) \\ \llbracket \text{if } b \text{ then } c_1 \text{ else } c_2 \rrbracket.(\text{dvg}) &\triangleq \text{test}(\llbracket b \rrbracket.(\text{tts})) \circ \llbracket c_1 \rrbracket.(\text{dvg}) \cup \text{test}(\llbracket b \rrbracket.(\text{ffs})) \circ \llbracket c_2 \rrbracket.(\text{dvg}). \end{aligned}$$

The diverging case of  $\llbracket c_1; c_2 \rrbracket$  means that either the execution of  $c_1$  diverges, or the execution of  $c_1$  terminates normally and then the execution of  $c_2$  diverges, which also indicates that the overloading of “ $\circ$ ” does make sense. Furthermore, the overloaded composition also enjoys the following common algebraic properties: for any binary relations  $R_1$  and  $R_2$ ,

$$\begin{aligned} (R_1 \circ R_2) \circ Y &= R_1 \circ (R_2 \circ Y) && \text{associative law} \\ R_1 \circ (Y_1 \cup Y_2) &= R_1 \circ Y_1 \cup R_1 \circ Y_2 && \text{left distributive law} \\ (R_1 \cup R_2) \circ Y &= R_1 \circ Y \cup R_2 \circ Y && \text{right distributive law} \end{aligned}$$

where  $Y$ , along with  $Y_1$  and  $Y_2$ , can be either a unary or a binary relation, and the distributive law can be applied to infinite unions as well.

As we can see, all the denotation for composite statements can be defined by semantic operators **test**, “ $\cup$ ”, and “ $\circ$ ”. Besides, the definition for loop statements is postponed to the next section, and those for atomic cases are independent of the unification of operators, so we omit them here.

### 2.3 Semantic Operators for Realistic Languages

The above relational style of defining denotational semantics can be naturally extended to realistic settings. Take Clight language as an example. For a given set  $B$ , let  $B^*$  denote the set of all finite sequences of elements in  $B$ , and  $B^\infty$  the set of all infinite sequences of elements in  $B$ . We then use  $\text{Denote}$  to represent the denotation of Clight statements where  $\text{event}$  denotes the set of system-call results (including input-output events in CompCert).

```
Record Denote: Type := {
  nrm: state → event* → state → Prop;  (* nrm ⊆ state × event* × state *)
  ...
  fin_dvg: state → event* → Prop;        (* fin_dvg ⊆ state × event* *)
  inf_dvg: state → event∞ → Prop;       (* inf_dvg ⊆ state × event∞ *)
}.
```

Compared to WHILE, a Clight program’s terminating behavior is extended to a ternary relation which additionally records the event trace generated during the execution of programs. In addition, diverging behavior is divided into finite and infinite parts for distinguishing behaviors with finite events and infinite events. Other omitted fields of `Denote` will be further discussed in §3.

Let  $X \subseteq \text{state}$ ,  $R \subseteq \text{state} \times \text{event}^* \times \text{state}$ , and  $W \subseteq \text{state} \times (\text{event}^* \cup \text{event}^\infty)$ . In order to define  $\llbracket s \rrbracket$  for a Clight statement  $s$  in the same way as WHILE, we at least need to overload the following set operators, where  $\text{nil}$  denotes the empty sequence and “ $\cdot$ ” denotes the concatenation of two sequences. Notably, these operators still conform to the algebraic properties listed in §2.2.

$$\begin{aligned} \text{test}(X) &\triangleq \{(\sigma, \tau, \sigma) \mid \sigma \in X \wedge \tau = \text{nil}\} \\ R \circ W &\triangleq \{(\sigma_1, \tau) \mid \exists \sigma_2 \tau_1 \tau_2, (\sigma_1, \tau_1, \sigma_2) \in R \wedge (\sigma_2, \tau_2) \in W \wedge \tau = \tau_1 \cdot \tau_2\} \\ R_1 \circ R_2 &\triangleq \{(\sigma_1, \tau, \sigma_3) \mid \exists \sigma_2 \tau_1 \tau_2, (\sigma_1, \tau_1, \sigma_2) \in R_1 \wedge (\sigma_2, \tau_2, \sigma_3) \in R_2 \wedge \tau = \tau_1 \cdot \tau_2\} \end{aligned}$$

Up to this point, we can see that the key to our semantic-operator unification lies in the unification of sequential composition whose differences come from two aspects: heterogeneous fields for various behaviors in the same language and heterogeneous denotations for different languages. Although there are various definitions of sequential composition on different occasions, fortunately, they all have common algebraic properties, which allows us to build a unified definition of these operators, and uniformly prove the algebraic properties they have. As a result, it will be more convenient to formally mechanize the semantics of programs and more extensible to support various program features. This will be further justified in the rest sections of this paper.

### 3 SEMANTICS OF WHILE LOOPS

So far, we have introduced how to define the denotational semantics of composite statements like if and sequential statements with set operators dealing with denotations composed of multiple sets. In this section we proceed to formulate the semantics of loop statements based on Park's approach [Park 1979]. It turns out that  $\llbracket \text{while } b \text{ do } c \rrbracket.(\text{nrm})$  is the least fixed point of  $f$  and  $\llbracket \text{while } b \text{ do } c \rrbracket.(\text{dvg})$  is the greatest fixed point of  $g$ , where

$$\begin{aligned} f(x) &= \text{test}(\llbracket b \rrbracket.(\text{ffs})) \cup (\text{test}(\llbracket b \rrbracket.(\text{tts})) \circ \llbracket c \rrbracket.(\text{nrm}) \circ x), \text{ and } x \subseteq \text{state} \times \text{state} \\ g(x) &= \text{test}(\llbracket b \rrbracket.(\text{tts})) \circ (\llbracket c \rrbracket.(\text{dvg}) \cup (\llbracket c \rrbracket.(\text{nrm}) \circ x)), \text{ and } x \subseteq \text{state} \end{aligned}$$

We construct these two fixed points using Kleene fixed point theorem and Knaster-Tarski fixed point theorem.

*Definition 3.1.* A poset (partially ordered set)  $(A, \leq)$  is a set  $A$  along with a reflexive, antisymmetric and transitive relation  $\leq$  on  $A$ . We refer to the standard notions of upper and lower bounds, least upper bound (lub) denoted by  $\sqcup$ , greatest lower bound (glb), monotonic functions, and continuous functions. Let  $(A, \leq_A)$  be a poset and  $\mathcal{P}(A)$  denotes the power-set of  $A$ . A chain  $S$  in  $A$  is a totally ordered subset of  $A$ , i.e.,  $\forall a, b \in S, a \leq_A b$  or  $b \leq_A a$ . If every chain in  $A$  has a least upper bound, then  $A$  is called a *complete partial ordering* (CPO).

**THEOREM 3.2 (KLEENE FIXED POINT).** *If poset  $(A, \leq_A)$  is a CPO and  $F : A \rightarrow A$  is monotonic and continuous, then  $F$  has a least fixed point  $\mu F$ . Let  $\perp$  be the least element of  $A$ , and then  $\mu F = \sqcup \{F^i(\perp) \mid i = 0, 1, 2, \dots\}$ .*

*Definition 3.3 (Complete lattice).* A poset  $(A, \leq_A)$  is a *complete lattice* if every subset  $S$  of  $A$  has a least upper bound and a greatest lower bound.

**THEOREM 3.4 (KNASTER-TARSKI THEOREM FOR GREATEST FIXED POINT).** *If poset  $(A, \leq_A)$  is a complete lattice and  $F : A \rightarrow A$  is monotonic, then  $F$  has a greatest fixed point  $\nu F$  and  $\nu F = \sqcup \{x \in A \mid x \leq_A F(x)\}$ .*

It's trivial to show that  $(\mathcal{P}(\text{state} \times \text{state}), \subseteq)$  is a CPO and  $f$  is monotonic and continuous on it, and  $(\mathcal{P}(\text{state}), \subseteq)$  is a complete lattice and  $g$  is a monotonic function on it. To this end, the semantics of the while statement in WHILE can be defined as follows:

$$\begin{aligned} \llbracket \text{while } b \text{ do } c \rrbracket.(\text{nrm}) &\triangleq \mu x. \text{test}(\llbracket b \rrbracket.(\text{ffs})) \cup (\text{test}(\llbracket b \rrbracket.(\text{tts})) \circ \llbracket c \rrbracket.(\text{nrm}) \circ x) \\ \llbracket \text{while } b \text{ do } c \rrbracket.(\text{dvg}) &\triangleq \nu x. \text{test}(\llbracket b \rrbracket.(\text{tts})) \circ (\llbracket c \rrbracket.(\text{dvg}) \cup (\llbracket c \rrbracket.(\text{nrm}) \circ x)) \end{aligned}$$

### 4 SEMANTICS OF PROCEDURE CALL

In this section, we use another toy language PCALL to demonstrate our key design when formalizing the semantics of a program with procedure calls. Specifically, a PCALL program consists of a list of procedures  $p_1, \dots, p_n$ . A procedure  $p$  is composed of procedure name  $i_p$  and procedure body

$c_p$  whose syntax is shown as follows.

$$c \triangleq \text{skip} \mid \text{atom} \mid \text{choice } c_1 \ c_2 \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid \text{call } i$$

Remark that the procedure call here does not have any arguments or return values for simplicity. The standard way [Back 1983; Park 1979] to define the denotational semantics of such a program is parameterizing the denotation of procedures with the behavior of invoked procedures, and then taking fixed points on recursive domain equations. We first summarize this approach in §4.1.

#### 4.1 Terminating Behavior of PCALL

Specifically, the terminating behavior of a PCALL program is defined in three steps: the semantics of statements, the semantics of procedures, and the semantics of the whole program.

Firstly, consider a semantic oracle  $\chi$  where  $\chi \subseteq \text{ident} \times \text{state} \times \text{state}$ , and  $\text{ident}$  is the set of procedure names such that an element  $(i, \sigma_0, \sigma_1) \in \chi$  if and only if executing the procedure named  $i$  from state  $\sigma_0$  could normally terminate at state  $\sigma_1$ . Then the terminating behavior of PCALL statements is a binary relation  $\llbracket c \rrbracket_{\chi \cdot (\text{nrm})}$ . It states that an element  $(\sigma_0, \sigma_1) \in \llbracket c \rrbracket_{\chi \cdot (\text{nrm})}$  if and only if executing statement  $c$  from state  $\sigma_0$  could terminate at state  $\sigma_1$ , in which the semantics of procedure calls will be given by  $\chi$ , i.e.,

$$\llbracket \text{call } i \rrbracket_{\chi \cdot (\text{nrm})} \triangleq \{(\sigma_0, \sigma_1) \mid (i, \sigma_0, \sigma_1) \in \chi\}$$

Besides, the semantics of other statements only need to be slightly modified based on WHILE. Take the sequential statement as an example:

$$\llbracket c_1; c_2 \rrbracket_{\chi \cdot (\text{nrm})} \triangleq \llbracket c_1 \rrbracket_{\chi \cdot (\text{nrm})} \circ \llbracket c_2 \rrbracket_{\chi \cdot (\text{nrm})}$$

Secondly, the terminating behavior of a procedure  $\llbracket p \rrbracket_{\chi \cdot (\text{nrm})}$  satisfies  $\llbracket p \rrbracket_{\chi \cdot (\text{nrm})} \subseteq \{i_p\} \times \text{state} \times \text{state}$ , and is defined by the semantics of its body  $\llbracket c_p \rrbracket_{\chi \cdot (\text{nrm})}$ , i.e.,

$$\llbracket p \rrbracket_{\chi \cdot (\text{nrm})} \triangleq \{(i_p, \sigma_0, \sigma_1) \mid (\sigma_0, \sigma_1) \in \llbracket c_p \rrbracket_{\chi \cdot (\text{nrm})}\}$$

Thus, an element  $(i_p, \sigma_0, \sigma_1) \in \llbracket p \rrbracket_{\chi \cdot (\text{nrm})}$  if and only if executing procedure  $p$  at state  $\sigma_0$  could eventually terminate at state  $\sigma_1$ . Finally, the terminating behavior of a program that contains a list of procedures  $p_1, \dots, p_n$  can be defined as:

$$\llbracket p_1; \dots; p_n \rrbracket_{\chi \cdot (\text{nrm})} \triangleq \mu\chi. (\llbracket p_1 \rrbracket_{\chi \cdot (\text{nrm})} \cup \dots \cup \llbracket p_n \rrbracket_{\chi \cdot (\text{nrm})})$$

By Kleene fixed point theorem, it says that a procedure call terminates if and only if there exists a natural number  $n$  such that the procedure call terminates with maximally  $n$  layers of nested calls. We proceed to describe how we extend this approach to support semantic linking of open modules in §4.2, and how we define the diverging behavior in §4.3.

#### 4.2 Semantics of Modules and Semantic Linking

In a more realistic setting, a program is composed of multiple modules, each of which contains a list of procedures and is compiled individually. We find that the traditional approach [Back 1983] (which we summarize in §4.1) can be easily extended to this setting by parameterizing the semantics of open modules with the terminating behavior of external procedures<sup>3</sup>.

<sup>3</sup>External procedures are those not defined in the current module.

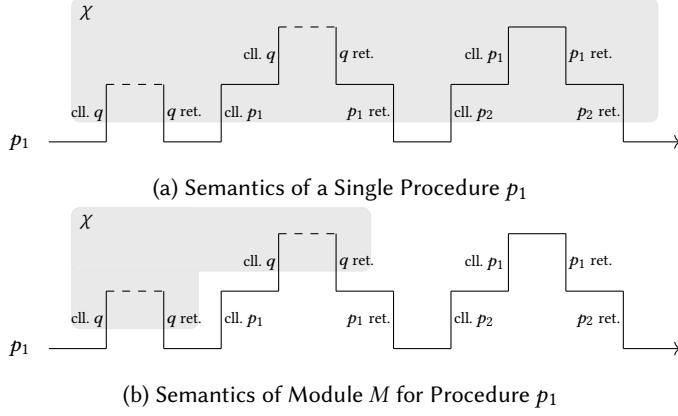


Fig. 3. Comparison Between the Semantics of Procedures and that of Modules. Here raising edges denote function calls (abbr. cll.), falling edges denote function returns (abbr. ret.), and the dashed line represents the behavior of external procedure  $q$ . The shaded areas show that for  $\llbracket p_1 \rrbracket_{\chi}^{(nrm)}$ , the behavior of every function call of  $p_1$  is interpreted by  $\chi$ , while for  $\llbracket M \rrbracket_{\chi}^{(nrm)}$ , only the behavior of external calls are interpreted by  $\chi$ .

**Semantics of open modules.** Formally, we use  $\llbracket M \rrbracket_{\chi}^{(nrm)}$  to represent the terminating behavior of an open module. It means that an element  $(i, \sigma_0, \sigma_1) \in \llbracket M \rrbracket_{\chi}^{(nrm)}$  if and only if there exists a procedure named  $i$  in module  $M$  and its execution beginning with state  $\sigma_0$  could finally terminate at state  $\sigma_1$ , in which the terminating behavior of external procedures is given by an oracle  $\chi$  (where  $\chi \subseteq \text{ident} \times \text{state} \times \text{state}$ ). Then the terminating behavior of an open module  $M$  that contains a list of procedures  $p_1, \dots, p_n$  can be defined as:

$$\llbracket M \rrbracket_{\chi}^{(nrm)} \triangleq \mu\chi_0. (\llbracket p_1 \rrbracket_{\chi_0 \cup \chi}^{(nrm)} \cup \dots \cup \llbracket p_n \rrbracket_{\chi_0 \cup \chi}^{(nrm)}), \text{ where } M = p_1; \dots; p_n \quad (2)$$

It's worth noting that although both the terminating behavior of a single procedure and that of an open module are subsets of  $\text{ident} \times \text{state} \times \text{state}$ , they differ in the following aspects:

- Depend on different semantic oracles. The semantics of a procedure depends on the semantics of all invoked procedures, even including recursive calls to itself. In comparison, the semantics of an open module depends on the semantics of external procedures only, as shown in Fig. 3;
- Accommodate different possible behaviors.  $\llbracket M \rrbracket_{\chi}^{(nrm)}$  can be seen as a function from the name of a procedure in module  $M$  to its terminating behavior, i.e., it interprets the terminating behavior of all procedures in module  $M$ , whereas  $\llbracket p \rrbracket_{\chi}^{(nrm)}$  only interprets the terminating behavior of the current procedure  $p$ .

**Semantic linking.** Semantic linking means composing the behavior of individual modules as a whole so that cross-module calls between module  $M_1$  and module  $M_2$  before linking become “internal” calls after linking. That is, after semantic linking, only external calls outside module  $M_1$  and module  $M_2$  need to be interpreted by a semantic oracle. Therefore, the definition of semantic linking is straightforward: assume that the terminating behavior of procedures outside module  $M_1$  and  $M_2$  are given by an oracle  $\chi$ , the terminating behavior of semantic linking is defined as:

$$(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi}^{(nrm)} \triangleq \mu\chi_0. (\llbracket M_1 \rrbracket_{\chi_0 \cup \chi}^{(nrm)} \cup \llbracket M_2 \rrbracket_{\chi_0 \cup \chi}^{(nrm)})$$

As we can see, similar to the semantic definition of modules, the semantic linking between two modules is defined by merging their denotations as if all the procedures reside in one module and

then taking the fixed point. Based on these definitions, the equivalence between semantic and syntactic linking is shown as follows.

**THEOREM 4.1 (EQUIVALENCE BETWEEN SEMANTIC AND SYNTACTIC LINKING).** *For any modules  $M_1$  and  $M_2$ ,  $\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket = \llbracket M_1 + M_2 \rrbracket$ , where “+” is syntactic linking, physically putting modules together.*

**PROOF.** The key to this proof relies on the following Lemma 4.2 about Kleene fixed point, and all the proofs have been formalized in Coq.  $\square$

**LEMMA 4.2 (COINCIDE THEOREM 1).** *Given CPOs  $A_1$  and  $A_2$ , for any monotonic and continuous functions  $f : A_1 \times A_2 \rightarrow A_1$  and  $g : A_1 \times A_2 \rightarrow A_2$ ,*

$$\mu(x, y).(\mu x_0.f(x_0, y), \mu y_0.g(x, y_0)) = \mu(x, y).(f(x, y), g(x, y))$$

In order to prove Thm. 4.1, we only need to instantiate the function  $f$  and  $g$  with the denotation of procedures within module  $M_1$  and module  $M_2$  respectively, which maps callee’s denotation to the caller’s denotation. On the left side of the lemma, the semantics of module  $M_1$  and module  $M_2$  are first derived individually, and then by the fixed point on the merged denotation of these two modules, the linked semantics is further derived, which corresponds to semantic linking. On the right side, we first merge the semantics of procedures in the two modules and then take the fixed point to obtain the whole semantics of the two modules, which corresponds to syntactic linking. The essence of Lemma 4.2 is illustrated by Fig. 4, which shows the two different ways of deriving fixed points will eventually converge to the same fixed point, and we refer to appendix B for mathematical proofs.

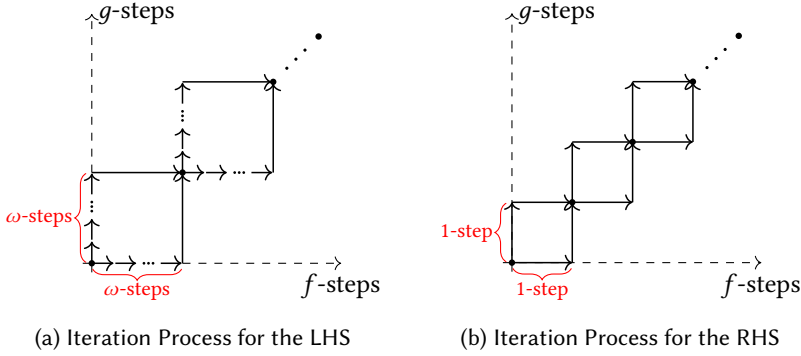


Fig. 4. An Intuitive Illustration of Lemma 4.2. The right-arrow denotes one step of iterations of  $f$  on the  $x$ -field with some fixed  $y$ , i.e.,  $(x, y) \mapsto (f(x, y), y)$ . Furthermore,  $\omega$ -steps of  $f$  is  $(x, y) \mapsto (\mu x_0.f(x_0, y), y)$ . The up-arrow denotes one step of iterations of  $g$  on the  $y$ -field with some fixed  $x$ , i.e.,  $(x, y) \mapsto (x, g(x, y))$ , and  $\omega$ -steps of  $g$  is  $(x, y) \mapsto (x, \mu y_0.g(x, y_0))$ .

### 4.3 Diverging Behavior of PCALL

The diverging behavior of a procedure can be induced either by its internal statements, or by recursive procedure calls. In this case, defining the diverging behavior of a procedure would be tricky, since the semantic oracle only tells us the terminating behavior of invoked procedures.

We solve this problem as follows: when defining  $\llbracket c \rrbracket_\chi$  and  $\llbracket p \rrbracket_\chi$ , we only consider internal divergence (caused by dead loops) first. Those caused by procedure calls (including dead loops in callee procedures and infinite layers of nested calls) are captured at the moment of defining  $\llbracket M \rrbracket_\chi$ . For this purpose, we further add the following set  $\text{c11}$  to  $\text{CDenote}$  to record the point of procedure calls,

so that an element  $(\sigma_0, (i, \sigma_1)) \in \llbracket c \rrbracket_{\chi} \cdot (\text{c11})$  if and only if executing statement  $c$  from state  $\sigma_0$  could eventually reach a point of procedure call named  $i$  at state  $\sigma_1$ .

$\text{c11}: \text{state} \rightarrow \text{call\_info} \rightarrow \text{Prop}$ , where  $\text{call\_info} \triangleq \text{id} \times \text{state}$ .

The calling behavior  $\llbracket c \rrbracket_{\chi} \cdot (\text{c11})$  is also recursively defined on the syntax of statements, for example,

$$\begin{aligned} \llbracket c_1; c_2 \rrbracket_{\chi} \cdot (\text{c11}) &\triangleq \llbracket c_1 \rrbracket_{\chi} \cdot (\text{c11}) \cup (\llbracket c_1 \rrbracket_{\chi} \cdot (\text{nrn}) \circ \llbracket c_2 \rrbracket_{\chi} \cdot (\text{c11})) \\ \llbracket \text{call } i \rrbracket_{\chi} \cdot (\text{c11}) &\triangleq \{(\sigma, (i, \sigma)) \mid \sigma \in \text{state}\} \end{aligned}$$

Correspondingly, the calling behavior of a procedure  $\llbracket p \rrbracket_{\chi} \cdot (\text{c11})$  is defined by its body, i.e.,

$$\llbracket p \rrbracket_{\chi} \cdot (\text{c11}) \triangleq \{(i_p, \sigma, \theta) \mid (\sigma, \theta) \in \llbracket c_p \rrbracket_{\chi} \cdot (\text{c11})\}$$

**Diverging behavior of open modules.** We use  $\llbracket M \rrbracket_{\chi} \cdot (\text{dvg})$  to denote the diverging behavior of open modules. Specifically,  $(i, \sigma) \in \llbracket M \rrbracket_{\chi} \cdot (\text{dvg})$  if and only if there exists a procedure named  $i$  in module  $M$  and its execution beginning with state  $\sigma$  could diverge due to internal statements after a finite number of procedure calls, or an infinite number of internal procedure calls inside module  $M$ . Thus, assume that the terminating behavior of external procedures is given by semantic oracle  $\chi$ , and let  $\hat{\chi} = \llbracket M \rrbracket_{\chi} \cdot (\text{nrn}) \cup \chi$  (i.e.,  $\hat{\chi}$  is the terminating behavior of all procedures, including both internals and externals),  $\llbracket M \rrbracket_{\chi} \cdot (\text{dvg})$  can be defined as:

$$\begin{aligned} \langle M \rangle_{\hat{\chi}} &\triangleq \llbracket p_1 \rrbracket_{\hat{\chi}} \cup \dots \cup \llbracket p_n \rrbracket_{\hat{\chi}} \\ \llbracket M \rrbracket_{\chi} \cdot (\text{dvg}) &\triangleq \nu \chi_0. \langle M \rangle_{\hat{\chi}} \cdot (\text{dvg}) \cup (\langle M \rangle_{\hat{\chi}} \cdot (\text{c11}) \circ \chi_0) \end{aligned}$$

where  $\langle M \rangle_{\hat{\chi}}$  is defined by simply merging corresponding behavior sets of procedures  $p_1, \dots, p_n$ .

We then use  $\llbracket M \rrbracket_{\chi} \cdot (\text{c11})$  ( $\subseteq \text{id} \times \text{state} \times \text{call\_info}$ ) to denote the calling behavior of a module. It means that an internal procedure of module  $M$  could eventually call an external procedure after a finite number of internal calls. Thus,  $\llbracket M \rrbracket_{\chi} \cdot (\text{c11})$  can be defined as:

$$\begin{aligned} \odot(M) &\triangleq \{(\theta, \theta) \mid \exists i_p \sigma, \theta = (i_p, \sigma) \wedge p \notin M\} \\ \llbracket M \rrbracket_{\chi} \cdot (\text{c11}) &\triangleq \mu \chi_0. (\langle M \rangle_{\hat{\chi}} \cdot (\text{c11}) \circ \odot(M)) \cup (\langle M \rangle_{\hat{\chi}} \cdot (\text{c11}) \circ \chi_0) \end{aligned}$$

**Diverging behavior of semantic linking.** Based on the definitions above, semantic linking can be easily extended to the  $\text{dvg}$  field and  $\text{c11}$  field: assume that the terminating behavior of external procedures outside  $M_1$  and  $M_2$  is given by semantic oracle  $\chi$ , and let  $\hat{\chi} = (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{nrn}) \cup \chi$ ,

$$\begin{aligned} (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{dvg}) &\triangleq \nu \chi_0. (\llbracket M_1 \rrbracket_{\hat{\chi}} \cdot (\text{dvg}) \cup \llbracket M_2 \rrbracket_{\hat{\chi}} \cdot (\text{dvg})) \cup \\ &\quad (\llbracket M_1 \rrbracket_{\hat{\chi}} \cdot (\text{c11}) \cup \llbracket M_2 \rrbracket_{\hat{\chi}} \cdot (\text{c11})) \circ \chi_0 \\ (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{c11}) &\triangleq \mu \chi_0. (\llbracket M_1 \rrbracket_{\hat{\chi}} \cdot (\text{c11}) \cup \llbracket M_2 \rrbracket_{\hat{\chi}} \cdot (\text{c11})) \circ \odot(M_1 + M_2) \cup \\ &\quad (\llbracket M_1 \rrbracket_{\hat{\chi}} \cdot (\text{c11}) \cup \llbracket M_2 \rrbracket_{\hat{\chi}} \cdot (\text{c11})) \circ \chi_0 \end{aligned}$$

**Remark.** There may exist traditional approaches that, for example, assume another semantic oracle  $\chi'$  ( $\subseteq \text{id} \times \text{state}$ ) to interpret the diverging behavior of procedure calls, so that the diverging behavior of a module can be defined by taking the greatest fixed point of corresponding recursive equations similar to formula (2). However, the approach proposed in this subsection has better extensibility, which will be further justified in §5.3.

## 5 SEMANTICS OF COMPCERT FRONT-END LANGUAGES

In this section, we focus on how to further extend the PCALL language to the CompCert front-end languages featured with C-like function calls, control flow, and divergence with event trace.

We will proceed in the following steps: we first extend procedure calls to C-like function calls in §5.1, then discuss how to support the control flow of CompCert front-end languages (take Clight

programs as an example) in §5.2, and finally define the diverging behavior of recursive structures with event trace in §5.3.

### 5.1 Semantics of C-like Function Calls

The method for handling procedure calls in PCALL can be easily extended to support C-like function calls which additionally take arguments and return values into account.

Likewise, we need an oracle  $\chi$  to denote the terminating behavior of callee functions, and in this setting  $\chi \subseteq \text{ident} \times \text{val}^* \times \text{state} \times \text{event}^* \times \text{state} \times \text{val}$  such that an element  $(i, \ell, \sigma, \tau, \sigma', r) \in \chi$  if and only if calling function  $i$  with arguments  $\ell$  at state  $\sigma$  will terminate at state  $\sigma'$  with a return value  $r$ , producing event trace  $\tau$ , where  $\text{val}$  is the set of values ranging over integers, floats, pointers, as well as an undefined value  $\text{Vundef}$ . Then for a function  $F$  that contains function name  $i_F$  and function body  $s_F$ , its terminating behavior  $\llbracket F \rrbracket_{\chi}^{(\text{nrm})}$  satisfies  $\llbracket F \rrbracket_{\chi}^{(\text{nrm})} \subseteq \{i_F\} \times \text{val}^* \times \text{state} \times \text{event}^* \times \text{state} \times \text{val}$ , and is defined by the semantics of its function body  $\llbracket s_F \rrbracket_{\chi}^{(\text{nrm})}$ . The way that how we define the semantics of modules and support semantic linking is the same as PCALL.

### 5.2 Semantics of Control Flow Constructs

**Clight.** The Clight language provides control flow constructs including **break**, **continue**, and **return** statements to prematurely terminate the normal execution of statements. To capture these behaviors, we add the following sets to  $\text{Denote}$ .

```
brk: state  $\rightarrow$  event*  $\rightarrow$  state  $\rightarrow$  Prop;
ctn: state  $\rightarrow$  event*  $\rightarrow$  state  $\rightarrow$  Prop;
rtn: state  $\rightarrow$  event*  $\rightarrow$  state  $\rightarrow$  val  $\rightarrow$  Prop.
```

It means that for any element  $(\sigma_0, \tau, \sigma_1)$  in  $\llbracket s \rrbracket_{\chi}^{(\text{brk})}$  or  $\llbracket s \rrbracket_{\chi}^{(\text{ctn})}$ , the execution of Clight statement  $s$  from  $\sigma_0$  will eventually reach state  $\sigma_1$  and then exit because of a **break** or **continue** statement, producing a sequence of input-output events  $\tau$ . The  $\text{rtn}$  set has a similar meaning but additionally with a return value.

$\llbracket \text{break} \rrbracket_{\chi}^{(\text{brk})} \triangleq \mathbb{1}$ , and other fields are assigned the empty set.

$\llbracket \text{continue} \rrbracket_{\chi}^{(\text{ctn})} \triangleq \mathbb{1}$ , and other fields are assigned the empty set.

Here  $\mathbb{1} \triangleq \{(\sigma, \text{nil}, \sigma) \mid \sigma \in \text{state}\}$ . The definitions above mean that once a **break** or **continue** statement is encountered, the execution will end prematurely, and the corresponding set  $\text{brk}$  or  $\text{ctn}$  records how it ends. The denotation of exiting by a **return** statement is defined similarly.

Clight uses one special loop (**loop**  $s_1$   $s_2$ ) to encode all the three kinds of C loops where (**loop**  $s_1$   $s_2$ ) means that statements  $s_1$  and  $s_2$  will be executed repeatedly in a sequential way, and a “continue” jump in  $s_1$  will branch to  $s_2$ . Then the **for** statement of C language for example can be defined as:

**for**( $s_1$ ;  $e$ ;  $s_2$ ) { $s_3$ }  $\triangleq s_1$ ; (**loop** (**if** ( $e$ ) **then skip else break**);  $s_3$   $s_2$ )

The terminating behavior of Clight loops satisfies the following equations.

$$N_1 = \llbracket s_1 \rrbracket_{\chi}^{(\text{nrm})} \cup \llbracket s_1 \rrbracket_{\chi}^{(\text{ctn})} \quad N_2 = N_1 \circ \llbracket s_2 \rrbracket_{\chi}^{(\text{nrm})} \quad (3)$$

$$\llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}^{(\text{nrm})} \triangleq \mu x. \llbracket s_1 \rrbracket_{\chi}^{(\text{brk})} \cup (N_1 \circ \llbracket s_2 \rrbracket_{\chi}^{(\text{brk})}) \cup (N_2 \circ x) \quad (4)$$

Here  $N_1$  denotes the behavior that loop body  $s_1$  either ends normally or prematurely due to a continue statement in  $s_1$ , and  $N_2$  denotes the behavior of a “sequential” execution of loop bodies  $s_1$  and  $s_2$ . Then, the execution of loops could normally terminate if  $s_1$  or  $s_2$  breaks the loop after executing the loop bodies a finite number of times.

**Csharpminor.** The control flow of Csharpminor is structured with the **block** and **exit** statement. For instance, the execution of statement  $\text{block} \{ \text{block} \{ s_1; \text{exit}(1) \}; s_2 \}; s_3$  is equivalent to the execution of  $s_1; s_3$  since the statement **exit**( $n$ ) terminates prematurely the execution of the ( $n+1$ ) layers of nested **block** statements. We then use a set  $\text{blk} (\subseteq \text{nat} \times \text{state} \times \text{event}^* \times \text{state})$  to capture this behavior so that an element  $(n, \sigma_0, \tau, \sigma_1) \in \llbracket u \rrbracket_{\chi}(\text{blk})$  if and only if executing the Csharpminor statement  $u$  from  $\sigma_0$  could terminate at  $\sigma_1$  and  $n$  records the layers of nested blocks to exit. Thus, the denotation of Csharpminor statement  $\llbracket u \rrbracket_{\chi}$  satisfies the following equations.

$$\begin{aligned} \llbracket \text{block}\{u\} \rrbracket_{\chi}(\text{norm}) &\triangleq \llbracket u \rrbracket_{\chi}(\text{norm}) \cup \llbracket u \rrbracket_{\chi}(\text{blk})_0 \\ \llbracket \text{block}\{u\} \rrbracket_{\chi}(\text{blk}) &\triangleq \{(n, \sigma, \tau, \sigma') \mid (n+1, \sigma, \tau, \sigma') \in \llbracket u \rrbracket_{\chi}(\text{blk})\} \\ \llbracket \text{exit}(n) \rrbracket_{\chi}(\text{blk}) &\triangleq \{(n_0, \sigma, \text{nil}, \sigma) \mid n_0 = n\}, \text{ other fields are assigned } \emptyset \end{aligned}$$

where  $(\text{blk})_n \triangleq \{(\sigma, \tau, \sigma') \mid (n, \sigma, \tau, \sigma') \in \text{blk}\}$  for any  $n \in \text{nat}$ . The above definitions mean that the **block** statement could normally terminate if its internal statement  $u$  terminates normally or exits one layer of block execution prematurely; and the **block** statement could early exit  $n$  layers if its internal body  $u$  early exits  $n+1$  layers of blocks; the other fields of  $\llbracket \text{block}\{u\} \rrbracket_{\chi}(\text{blk})$  are directly determined by the corresponding set of the internal statement  $u$ . Simultaneously, when an **exit** statement is encountered, the later statements will no longer execute normally and the layer of blocks to be early exited is recorded by the  $\text{blk}$  set, and other fields of the **exit** statement are assigned the empty set.

In summary, we add two sets  $\text{brk}$  and  $\text{ctn}$  to  $\text{Denote}$ , and add one set  $\text{blk}$  to  $\text{Denote}$  respectively so as to cope with control flow constructs for Clight and Csharpminor. For ease of distinction, we use  $\text{Clit.Denote}$  and  $\text{Cshm.Denote}$  to distinguish these two denotations. Additionally, we add a set  $\text{err} \subseteq \text{state} \times \text{event}^*$  to them to capture the aborting behavior of programs, which can be seen as a kind of terminating behavior and is even simpler to define.

### 5.3 Divergence with Event Trace

When event traces are taken into account, handling the divergence of recursive structures (e.g., Clight loops) can be tricky. If simply using the greatest fixed point discussed in §3 to define the diverging behavior of loops, then we may obtain more behaviors than what we expect, for example,

$$\llbracket \text{while true do skip} \rrbracket_{\chi}(\text{dvg}) = \nu x. \mathbb{1} \circ (\emptyset \cup \mathbb{1} \circ x) = \nu x. x = \text{state} \times (\text{event}^* \cup \text{event}^{\infty})$$

This is wrong! Since executing  $(\text{while true do skip})$  should only generate the empty event trace. The problem here is similar to the classic *stuttering problem* [Leroy 2009b] when defining an operational-semantics-based simulation relation.

**Our solution.** We use silent operator  $\Delta$  and non-silent operator  $\blacktriangle$  to explicitly filter silent and non-silent event trace when defining the diverging behavior of recursive structures as follows, where  $R \subseteq A \times B^* \times A$  and  $X \subseteq A \times B^*$  for given sets  $A, B$ .

$$\begin{aligned} \blacktriangle R &\triangleq \{(\sigma, \tau, \sigma') \mid (\sigma, \tau, \sigma') \in R \wedge \tau \neq \text{nil}\} \\ \Delta R &\triangleq \{(\sigma, \tau, \sigma') \mid (\sigma, \tau, \sigma') \in R \wedge \tau = \text{nil}\} \\ \Delta X &\triangleq \{(\sigma, \tau) \mid (\sigma, \tau) \in X \wedge \tau = \text{nil}\} \end{aligned}$$

Recall that we use  $\llbracket s \rrbracket_{\chi}(\text{fin\_dvg})$  and  $\llbracket s \rrbracket_{\chi}(\text{inf\_dvg})$  to denote the diverging behavior with finite and infinite event traces generated by executing statement  $s$ , respectively. Then the silently diverging behavior of Clight loops can be defined as follows, where  $N_1$  and  $N_2$  are defined by formula (3).

$$\llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}(\text{fin\_dvg}) \triangleq \mu x. \llbracket s_1 \rrbracket_{\chi}(\text{fin\_dvg}) \quad (5)$$

$$\cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}(\text{fin\_dvg}) \quad (6)$$

$$\cup \Delta (vx_0.N_2 \circ x_0) \quad (7)$$

$$\cup N_2 \circ x \quad (8)$$

It indicates that a Clight loop could silently diverge since the loop body  $s_1$  does (5), or the loop body  $s_2$  does (6), or the entire loop itself silently diverges (7), or it belongs to one of the above three cases after being executed a finite number of times (8).

The non-silently diverging behavior (i.e, reacting behavior) of Clight loops is defined as follows.

$$\begin{aligned} \llbracket \text{loop } s_1 s_2 \rrbracket_{\chi}.(\text{inf\_dvg}) &\triangleq vx.\mathbb{B} \circ \mathbb{D} \cup (\mathbb{B} \circ \blacktriangle N_2) \circ x \\ \text{where } \mathbb{B} &= \mu x. \mathbb{1} \cup \Delta N_2 \circ x, \text{ and} \\ \mathbb{D} &= \llbracket s_1 \rrbracket_{\chi}.(\text{inf\_dvg}) \cup (N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{inf\_dvg})) \end{aligned}$$

Here,  $\mathbb{B}$  denotes the behavior of silently executing the loop body a finite number of times, and  $\mathbb{D}$  denotes the non-silently diverging behavior when executing loop body  $s_1$  or  $s_2$ . It indicates that either the loop body diverges non-silently (i.e., reacts) after silently executing it a finite number of times (i.e.,  $\mathbb{B} \circ \mathbb{D}$ ), or one observable behavior must be triggered after silently executing the loop body a finite number of times (i.e.,  $\mathbb{B} \circ \blacktriangle N_2$ ) and then this process repeats infinitely.

We can similarly define  $\llbracket M \rrbracket_{\chi}.(\text{fin\_dvg})$  and  $\llbracket M \rrbracket_{\chi}.(\text{inf\_dvg})$  like how we handle loops. The key point is to use  $\Delta \llbracket M \rrbracket_{\chi}.(\text{c11})$  and  $\blacktriangle \llbracket M \rrbracket_{\chi}.(\text{c11})$  in the definition so that we can distinguish the cases that zero events happen before a call and the cases that at least one event happens before the call.

**Remark.** When defining  $\llbracket M \rrbracket_{\chi}.(\text{fin\_dvg})$  and  $\llbracket M \rrbracket_{\chi}.(\text{inf\_dvg})$ , the approach to using another semantic oracle  $\chi'$  to interpret the diverging behavior of callee functions remarked in §4.3 is not available here, since silent divergence and non-silent divergence, as illustrated at the beginning of this subsection, can not be defined by trivially taking the greatest fixed point.

## 6 BEHAVIOR REFINEMENT

Compilation correctness is ubiquitously formulated as a behavior refinement relation which states that every behavior of the target program is one possible behavior of the source. In this section, we start from typical examples of behavior refinements (§6.1) and then introduce a kind of refinement algebra capable of covering all these cases (§6.2). We end this section by demonstrating that our framework can easily support module-level compositionality (§6.3).

### 6.1 Examples of Behavior Refinement

*Example 6.1 (Behavior refinement for simple transformation).* Consider simple transformations like removing sequenced **skip** statements, or making sequenced sequential statements (e.g.,  $(c_1; c_2); c_3$ ) right-associative. Such program transformation is a function  $\mathcal{T}$  that turns a program  $c$  into another program  $\mathcal{T}(c)$  in the same language, where the initial (ending) states before and after the transformation are unchanged. In this case, the state-matching relation in the refinement diagram (as Fig. 1a shows) can be the identity relation. Therefore, if we consider such program transformation, behavior refinement  $\llbracket \mathcal{T}(c) \rrbracket \sqsubseteq \llbracket c \rrbracket$  can be defined as:

$$\llbracket \mathcal{T}(c) \rrbracket.(\text{nrm}) \subseteq \llbracket c \rrbracket.(\text{nrm}) \text{ and } \llbracket \mathcal{T}(c) \rrbracket.(\text{dvg}) \subseteq \llbracket c \rrbracket.(\text{dvg})$$

*Example 6.2 (Behavior refinement for original CompCert).* In CompCert, behavior refinement is originally described as simulation diagrams for small-step semantics, which can be adapted to our denotation-based setting shown in Fig. 5, where  $\tau_0 \leq_T \tau$  means that  $\tau_0$  is the prefix of  $\tau$ .

Fig. 5 says that for every terminating (or diverging) behavior of the target program, there will exist consistent behavior in the source end, or the source program aborts (denoted as  $\perp$ ) at some middle point. This definition is valid since compilation correctness implies that once the source

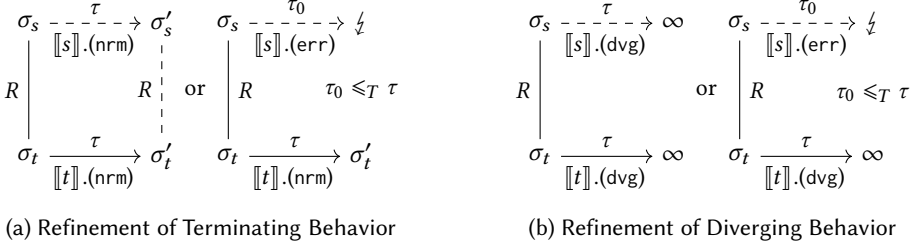


Fig. 5. Behavior Refinement for Original CompCert

program may abort, the compiled target program could do anything unpredictable. In this case, behavior refinement  $\llbracket t \rrbracket \sqsubseteq \llbracket s \rrbracket$  can be formally defined as: for a given state-matching relation  $R$ ,

$$\begin{aligned}
 & \forall \sigma_t \tau \sigma'_t \sigma_s, (\sigma_t, \tau, \sigma'_t) \in \llbracket t \rrbracket.(nrm) \Rightarrow (\sigma_s, \sigma_t) \in R \Rightarrow \\
 & \quad (\exists \sigma'_s, (\sigma_s, \tau, \sigma'_s) \in \llbracket s \rrbracket.(nrm) \wedge (\sigma'_s, \sigma'_t) \in R) \vee \\
 & \quad (\exists \tau_0, (\sigma_s, \tau_0) \in \llbracket s \rrbracket.(err) \wedge \tau_0 \leq_T \tau); \text{ and} \\
 & \forall \sigma_t \tau \sigma_s, (\sigma_t, \tau) \in \llbracket t \rrbracket.(dvg) \Rightarrow (\sigma_s, \sigma_t) \in R \Rightarrow \\
 & \quad (\sigma_s, \tau) \in \llbracket s \rrbracket.(dvg) \vee (\exists \tau_0, (\sigma_s, \tau_0) \in \llbracket s \rrbracket.(err) \wedge \tau_0 \leq_T \tau).
 \end{aligned}$$

As is well-known, the verification results of original CompCert are limited to separate compilation by the same compiler. In order to support horizontal compositionality between heterogeneous languages (e.g., C and assembly), a series of derivative works after CompCert, such as CompComp [Stewart et al. 2015] and CompCertM [Song et al. 2020], adopt open simulations to establish correspondence between the interaction semantics of the source and the target program. Latest findings made by CompCertO [Koenig and Shao 2021] and Zhang et al. [2023] propose a kind of Kripke Logic Relation to establish correspondences between open module interactions. We interpret their approach into our denotation-based framework as follows.

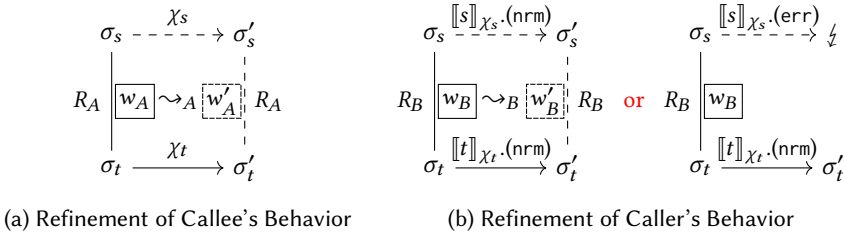


Fig. 6. Behavior Refinement for Open Modules. Here  $R_A$  and  $R_B$  are Kripke relations used for relating callee's behavior and caller's behavior respectively. Semantic oracle  $\chi_s$  and  $\chi_t$  are respectively used for interpreting function calls of the source program and that of the target program.

*Example 6.3 (Behavior refinement for open modules).* Given sets  $A_1, A_2$ , and  $W$ , a Kripke relation  $R : W \rightarrow \{X \mid X \subseteq A_1 \times A_2\}$  is a family of relations indexed by the set of *Kripke worlds*  $W$ , written as  $\mathcal{K}_W(A_1, A_2)$ . As the program executes from an initial state to some termination state, the Kripke worlds (e.g., memory injections) may evolve from a beginning world  $w$  to an ending world  $w'$  governed by an accessibility relation  $\sim$  (written as  $w \sim w'$ ). Such evolution is used to formulate rely-guarantee reasoning, which is essential for achieving horizontal compositionality.

As shown in Fig. 6, the evolution of worlds in the refinement of callee's behavior provides rely conditions for the refinement of caller's behavior. In turn, by assuming  $w_A \rightsquigarrow_A w'_A$ , the evolution of worlds in the refinement of caller's behavior should respect the guarantee condition  $w_B \rightsquigarrow_B w'_B$ . For instance, the caller's behavior refinement  $\llbracket t \rrbracket_{\chi_t} \sqsubseteq_{\text{nrn}} \llbracket s \rrbracket_{\chi_s}$  (for terminating case) can then be formally defined as:

$$\begin{aligned} \forall w_B \sigma_t \tau \sigma'_t \sigma_s, (\sigma_t, \tau, \sigma'_t) \in \llbracket t \rrbracket_{\chi_t} \cdot (\text{nrn}) \Rightarrow (\sigma_s, \sigma_t) \in R_B(w_B) \Rightarrow \\ (\exists w'_B \sigma'_s, (\sigma_s, \tau, \sigma'_s) \in \llbracket s \rrbracket_{\chi_s} \cdot (\text{nrn}) \wedge w_B \rightsquigarrow_B w'_B \wedge (\sigma'_s, \sigma'_t) \in R_B(w'_B)) \vee \\ (\exists \tau_0, (\sigma_s, \tau_0) \in \llbracket s \rrbracket_{\chi_s} \cdot (\text{err}) \wedge \tau_0 \leq_T \tau) \end{aligned}$$

## 6.2 Refinement Algebra

*Definition 6.4 (Refinement algebra).* A refinement algebra is a tuple  $(\mathcal{N}, \mathcal{E}, \mathcal{D}, \cup, \circ, \gamma)$ , where

- $\mathcal{N}, \mathcal{E}$  and  $\mathcal{D}$  are behavior sets. For instance,  $\mathcal{N}$  and  $\mathcal{D}$  are the set of terminating behaviors for the source and the target program respectively, and  $\mathcal{E}$  is the set of aborting behaviors.
- $\cup$  and  $\circ$  are semantic operators whose instances are introduced in §2. Recall that  $\cup$  is used for merging two behavior sets, and  $\circ$  for the composition of them.
- $\gamma : \mathcal{N} \times \mathcal{E} \rightarrow \mathcal{D}$  is a gamma function which, for instance, maps the pair of source program's behavior sets to the target program's behavior set.
- Semantic operators and the gamma function conform to the following algebraic properties:

$$\forall N_1 N_2 E_1 E_2, N_1 \subseteq N_2 \Rightarrow E_1 \subseteq E_2 \Rightarrow \gamma(N_1, E_1) \subseteq \gamma(N_2, E_2) \quad (9)$$

$$\forall N_1 N_2 E_1 E_2, N_1 = N_2 \Rightarrow E_1 = E_2 \Rightarrow \gamma(N_1, E_1) = \gamma(N_2, E_2) \quad (10)$$

$$\forall N_1 N_2 E_1 E_2, \gamma(N_1, E_1) \cup \gamma(N_2, E_2) \subseteq \gamma(N_1 \cup N_2, E_1 \cup E_2) \quad (11)$$

$$\forall N M E_1 E_2, \gamma(N, E_1) \circ \gamma(M, E_2) \subseteq \gamma(N \circ M, E_1 \cup N \circ E_2) \quad (12)$$

$$\begin{aligned} \forall N M E_1 E_2, \mu x. \gamma(M, E_2) \cup \gamma(N, E_1) \circ x \subseteq \\ \gamma(\mu x. M \cup N \circ x, \mu x. (E_1 \cup E_2) \cup N \circ x) \end{aligned} \quad (13)$$

$$\begin{aligned} \forall N M E_1 E_2, \nu x. \gamma(M, E_2) \cup \gamma(N, E_1) \circ x \subseteq \\ \gamma(\nu x. M \cup N \circ x, \mu x. (E_1 \cup E_2) \cup N \circ x) \end{aligned} \quad (14)$$

Properties (9) and (10) indicate the gamma function are monotonic and congruent; properties (11) ~ (14) mean that the gamma function is homomorphic in terms of the union operator, the composition operator, the least and greatest fixed points, which are respectively named as the union-inclusion property, the composition-inclusion property, the least- and greatest-fixed-point-inclusion property. Besides, for behaviors with event trace, the gamma function should also enjoy the silent inclusion and non-silent inclusion properties shown as follows.

$$\forall N E, \Delta \gamma(N, E) \subseteq \gamma(\Delta N, E) \quad (15)$$

$$\forall N E, \blacktriangle \gamma(N, E) \subseteq \gamma(\blacktriangle N, E) \quad (16)$$

**Instances of refinement algebra.** We then explain how various behavior refinements shown in §6.1 can be defined through the refinement algebra. The key to this is to find an appropriate definition of gamma that makes all the algebraic properties hold. It's worth noting that these algebraic properties can extremely help us to prove behavior refinement in an algebraically structured way.

For Example 6.1, behavior sets<sup>4</sup>  $\mathcal{N}$  and  $\mathcal{D}$  can be  $\mathcal{P}(\text{state} \times \text{state})$ , and  $\gamma(N_1, E_1) \triangleq N_1$ . Then the behavior refinement  $\llbracket \mathcal{T}(c) \rrbracket \sqsubseteq \llbracket c \rrbracket$  in Example 6.1 is reinterpreted as:

$$\llbracket \mathcal{T}(c) \rrbracket.(\text{nrn}) \subseteq \gamma(\llbracket c \rrbracket.(\text{nrn}), \emptyset) \text{ and } \llbracket \mathcal{T}(c) \rrbracket.(\text{dvg}) \subseteq \gamma(\llbracket c \rrbracket.(\text{dvg}), \emptyset)$$

To interpret the behavior refinement in Example 6.3, we define the gamma function as follows. Let  $\text{state}_s$  and  $\text{state}_t$  be the set of source program states and that of target program states respectively. For given sets  $N_s \subseteq \text{state}_s \times \text{event}^* \times \text{state}_s$  and  $E_s \subseteq \text{state}_s \times \text{event}^*$ ,

$$\begin{aligned} \forall(\sigma_t, \tau, \sigma'_t) \in \gamma(N_s, E_s) \text{ iff } \forall w \ \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow \\ (\exists w' \ \sigma'_s, (\sigma_s, \tau, \sigma'_s) \in N_s \wedge w \rightsquigarrow_W w' \wedge (\sigma'_s, \sigma'_t) \in R(w')) \vee \\ (\exists \tau_0, (\sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau). \end{aligned}$$

Here the gamma instance is parameterized with a Kripke relation  $\mathcal{K}_W(\text{state}_s, \text{state}_t)$  for relating the source and the target program states. We have formally proved in Coq that the gamma function defined here satisfies all the algebraic properties listed above. Then the behavior refinement  $\llbracket t \rrbracket_{\chi_t} \sqsubseteq_{\text{nrn}} \llbracket s \rrbracket_{\chi_s}$  shown in Fig. 6b can be redefined as:

$$\llbracket t \rrbracket_{\chi_t}.(\text{nrn}) \subseteq \gamma(\llbracket s \rrbracket_{\chi_s}.(\text{nrn}), \llbracket s \rrbracket_{\chi_s}.(\text{err}))$$

The refinement of other behaviors (e.g., the diverging behavior) shown in §5 can be interpreted with the refinement algebra in a similar way, based on which, we next show the refinement of denotations for CompCert front-end languages.

*Definition 6.5 (Refinement between statement denotations).* Given natural numbers<sup>5</sup>  $n_b, n_c$  and the aborting behavior of external functions  $\chi_e \subseteq \text{call\_info} \times \text{event}^*$ . A denotation  $D_2$  of  $\text{Cshm.Denote}$  is said to be a refinement of a denotation  $D_1$  of  $\text{Clit.Denote}$ , written as  $D_2 \lesssim_{(n_b, n_c, \chi_e)} D_1$  if and only if:

$$\begin{aligned} D_2.(\text{nrn}) &\subseteq \gamma(D_1.(\text{nrn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{blk})_{n_b} &\subseteq \gamma(D_1.(\text{brk}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{blk})_{n_c} &\subseteq \gamma(D_1.(\text{ctn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{fin\_dvg}) &\subseteq \gamma(D_1.(\text{fin\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{inf\_dvg}) &\subseteq \gamma(D_1.(\text{inf\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \end{aligned}$$

Here, we mainly list the refinement for terminating behaviors, control-flow-related behaviors, and diverging behaviors of Clight and Csharpminor statements for simplicity. In addition, for the translation of Csharpminor statements to Cminor statements, the refinement between a denotation  $D_3$  of  $\text{Cmin.Denote}$  and  $D_2$  of  $\text{Cshm.Denote}$  is defined similarly (see appendix D for details).

The definition of refinement between function denotations shown as follows is similar to the definition of refinement between statement denotations.

*Definition 6.6 (Refinement between function denotations).* Let  $\Phi_1$  and  $\Phi_2$  be the denotation of source functions and that of target functions respectively.  $\Phi_2$  is said to be a refinement of  $\Phi_1$ , written as  $\Phi_2 \sqsubseteq \Phi_1$  if and only if for any  $\chi_t, \chi_n$  and  $\chi_e$  such that  $\chi_t \subseteq \gamma(\chi_n, \chi_e)$ , the following conditions hold, where  $D_1 = (\Phi_1)_{\chi_s}$  and  $D_2 = (\Phi_2)_{\chi_t}$ :

$$\begin{aligned} D_2.(\text{dom}) &= D_1.(\text{dom}), \text{ where dom is the set of function names} \\ D_2.(\text{nrn}) &\subseteq \gamma(D_1.(\text{nrn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \end{aligned}$$

<sup>4</sup>Here behavior set  $\mathcal{E}$  can be ignored, since aborting behavior is not considered in this trivial example.

<sup>5</sup>Natural numbers  $n_b$  and  $n_c$  are arguments of the CSharpMinorGen transformation in CompCert, which are used to count the exiting number from nested blocks (i.e., **exit**  $n_b$  and **exit**  $n_c$  compiled from **break** and **continue** respectively).

$$\begin{aligned}
D_2.(\text{fin\_dvg}) &\subseteq \gamma(D_1.(\text{fin\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\
D_2.(\text{inf\_dvg}) &\subseteq \gamma(D_1.(\text{inf\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e)
\end{aligned}$$

### 6.3 Module-level Compositionality

Based on the definition of refinement between function denotations, we then have the following results, where Lemma 6.7 helps derive the refinement of modules from the refinement of functions, Thm. 6.8 is used to compose the correctness of each compilation phase, and Thm. 6.9 shows the correctness of module-level compositionality.

Specifically, in our implementation, we follow Koenig et al.'s approach [Koenig and Shao 2021; Zhang et al. 2023] to support vertical compositionality and use denotation-based semantic linking along with fixed-point-related theorems to support horizontal compositionality.

LEMMA 6.7. *For any modules  $M_s$  and  $M_t$ , if  $\llbracket M_t \rrbracket \sqsubseteq \llbracket M_s \rrbracket$ , then  $\llbracket M_t \rrbracket \sqsubseteq \llbracket M_s \rrbracket$ .*

THEOREM 6.8 (VERTICAL COMPOSITIONALITY). *For any modules  $M_1$ ,  $M_2$  and  $M_3$ , if  $\llbracket M_1 \rrbracket \sqsubseteq \llbracket M_2 \rrbracket$  and  $\llbracket M_2 \rrbracket \sqsubseteq \llbracket M_3 \rrbracket$ , then  $\llbracket M_1 \rrbracket \sqsubseteq \llbracket M_3 \rrbracket$ .*

THEOREM 6.9 (HORIZONTAL COMPOSITIONALITY). *For any modules  $T_1$ ,  $T_2$ ,  $S_1$  and  $S_2$ , if  $\llbracket T_1 \rrbracket \sqsubseteq \llbracket S_1 \rrbracket$  and  $\llbracket T_2 \rrbracket \sqsubseteq \llbracket S_2 \rrbracket$ , then  $\llbracket T_1 \rrbracket \oplus \llbracket T_2 \rrbracket \sqsubseteq \llbracket S_1 \rrbracket \oplus \llbracket S_2 \rrbracket$ .*

Finally, by the horizontal compositionality (Thm. 6.9) and the equivalence between semantic linking and syntactic linking (Thm. 4.1), we have the following separate compilation correctness.

COROLLARY 6.10 (SEPARATE COMPILATION). *For any modules  $T_1$ ,  $T_2$ ,  $S_1$  and  $S_2$ , if  $\llbracket T_1 \rrbracket \sqsubseteq \llbracket S_1 \rrbracket$  and  $\llbracket T_2 \rrbracket \sqsubseteq \llbracket S_2 \rrbracket$ , then  $\llbracket T_1 + T_2 \rrbracket \sqsubseteq \llbracket S_1 + S_2 \rrbracket$ .*

## 7 COMPILATION CORRECTNESS

**Translation Correctness from Clight to Csharpminor.** Let  $\mathcal{T}_1$  be the partial function translating a Clight statement to a Csharpminor statement (a translation example is shown in Fig. 2) and  $\mathcal{T}_2$  translating a Clight function to a Csharpminor function. Then we have the following results.

LEMMA 7.1. *For any natural numbers<sup>6</sup>  $n_b, n_c$  such that  $n_b \neq n_c$ , and behaviors  $\chi_t, \chi_s, \chi_e$  such that  $\chi_t \subseteq \gamma(\chi_s, \chi_e)$ , and for any Clight statement  $s$  and Csharpminor statement  $u$ ,*

$$\text{if } \mathcal{T}_1(n_b, n_c, s) = \text{succed}(u), \text{ then } \llbracket u \rrbracket_{\chi_t} \lesssim_{(n_b, b_c, \chi_e)} \llbracket s \rrbracket_{\chi_s}$$

PROOF. By taking induction on statement  $s$ , we obtain proof obligations asserting that for every syntactic construct of  $s$ , if each of its substructures satisfies the statement refinement relation with the compiled one, so does statement  $s$ .

As an illustration, we show one of the interesting proof obligations—when  $s$  is a sequential statement  $(s_1; s_2)$ . Since  $\mathcal{T}_1(n_b, n_c, s_1; s_2) = \text{succed}(u)$ , then  $\mathcal{T}_1(n_b, n_c, s_1) = \text{succed}(u_1)$  and  $\mathcal{T}_1(n_b, n_c, s_2) = \text{succed}(u_2)$ . The proof obligation is to show:

$$\begin{aligned}
\llbracket u_1 \rrbracket_{\chi_t} \lesssim_{(n_b, b_c, \chi_e)} \llbracket s_1 \rrbracket_{\chi_s} &\Rightarrow \llbracket u_2 \rrbracket_{\chi_t} \lesssim_{(n_b, b_c, \chi_e)} \llbracket s_2 \rrbracket_{\chi_s} \Rightarrow \\
&\llbracket u_1; u_2 \rrbracket_{\chi_t} \lesssim_{(n_b, b_c, \chi_e)} \llbracket s_1; s_2 \rrbracket_{\chi_s}
\end{aligned}$$

For simplicity, let  $N_i$ ,  $E_i$  and  $C_i$  respectively represent  $\llbracket s_i \rrbracket_{\chi_s}.(\text{nm})$ ,  $\llbracket s_i \rrbracket_{\chi_s}.(\text{err})$  and  $\llbracket s_i \rrbracket_{\chi_s}.(\text{c11})$  for  $i = 1, 2$ . Then according to Def. 6.5, the terminating case is to show:

$$\llbracket u_1; u_2 \rrbracket_{\chi_t}.(\text{nm}) = \llbracket u_1 \rrbracket_{\chi_t}.(\text{nm}) \circ \llbracket u_2 \rrbracket_{\chi_t}.(\text{nm}) \quad (17)$$

$$\subseteq \gamma(N_1, E_1 \cup C_1 \circ \chi_e) \circ \gamma(N_2, E_2 \cup C_2 \circ \chi_e) \quad (18)$$

<sup>6</sup>Clight statements **break** and **continue** are compiled into Csharpminor's **(exit  $n_b$ )** and **(exit  $n_c$ )** respectively.

$$\subseteq \gamma(N_1 \circ N_2, (E_1 \cup C_1 \circ \chi_e) \cup N_1 \circ (E_2 \cup C_2 \circ \chi_e)) \quad (19)$$

$$\subseteq \gamma(N_1 \circ N_2, (E_1 \cup N_1 \circ E_2) \cup (C_1 \cup N_1 \circ C_2) \circ \chi_e) \quad (20)$$

$$= \gamma(\llbracket s_1; s_2 \rrbracket_{\chi_s} \cdot (\text{norm}), \llbracket s_1; s_2 \rrbracket_{\chi_s} \cdot (\text{err}) \cup \llbracket s_1; s_2 \rrbracket_{\chi_s} \cdot (\text{c11}) \circ \chi_e) \quad (21)$$

The equivalence in (17) and between (20) ~ (21) are deduced by definition. The inclusion between (17) and (18) is deduced by the first and second induction hypotheses. The inclusion between (18) and (19) is deduced by the composition-inclusion property of the gamma function. The inclusion between (19) and (20) is deduced by the associative and distributive law of sequential composition and the commutative law of set union. Thus, the refinement of terminating behavior is proved. The refinement of diverging behavior for sequential statements can be proved in a similar way where the union inclusion of the gamma function will be used additionally.

It's worth noting that when  $s$  is an atomic statement (e.g., an assignment statement), the original CompCert proof is reused. Other proof obligations for the induction like the refinement for loop statements are also proved by the same technique, where properties including the least- and greatest-fixed-point-inclusion, silent and non-silent inclusion of the gamma function will be used.  $\square$

**THEOREM 7.2.** *For any Clight function  $F_s$  and Csharpminor function  $F_u$ ,*

$$\text{if } \mathcal{T}_2(F_s) = \text{succed}(F_u), \text{ then } \llbracket F_u \rrbracket \sqsubseteq \llbracket F_s \rrbracket$$

Remark that Thm. 7.2 is deduced from the translation correctness of Clight statements (i.e., Lemma 7.1), and the original CompCert proof is reused for verifying the translation of initialization when entering the function and the translation of memory freeing when leaving the function.

**Translation Correctness from Csharpminor to Cminor.** Let  $\mathcal{T}'_2$  be the partial function of CompCert translating a Csharpminor function to a Cminor function. Similarly, we have the following results for the second compilation phase.

**THEOREM 7.3.** *For any Csharpminor function  $F_u$  and Cminor function  $F_t$ ,*

$$\text{if } \mathcal{T}'_2(F_u) = \text{succed}(F_t), \text{ then } \llbracket F_t \rrbracket \sqsubseteq \llbracket F_u \rrbracket$$

**Summary.** In our framework, almost all the proofs (including proofs of all the theorems in §6.3 and in this section) have the following main proof steps (as shown in the proof of Lemma 7.1):

- By gamma's algebraic properties listed in (9) ~ (14), the behavior refinements of substructures are composed into an overall refinement, without unfolding the concrete definition of gamma (except for proofs for atomic statements);
- By the properties of semantic operators  $\cup$ ,  $\circ$ , and fixed points, proofs are finished in a purely relational subsystem like the deduction from (19) to (20) in Lemma 7.1.

Finally, by Thm. 7.2, Thm. 7.3, Thm. 6.9 and Thm. 4.1, our final theorem is shown as follows:

**THEOREM 7.4 (FINAL THEOREM).** *For any Clight module  $S_1, \dots, S_n$  and Cminor module  $T_1, \dots, T_n$ , if  $T_i$  is successfully compiled from  $S_i$  by the CompCert front-end for each  $i$ , then*

$$\llbracket T_1 + \dots + T_n \rrbracket \sqsubseteq \llbracket S_1 \rrbracket \oplus \dots \oplus \llbracket S_n \rrbracket$$

## 8 COMPARISON WITH PREVIOUS WORK

Compositional compiler verification has been an important research topic for a long time, as evidenced by extensive research efforts such as Beringer et al. [2014], Ramananandro et al. [2015], Stewart et al. [2015], Song et al. [2020], Koenig and Shao [2021], Zhang et al. [2023].

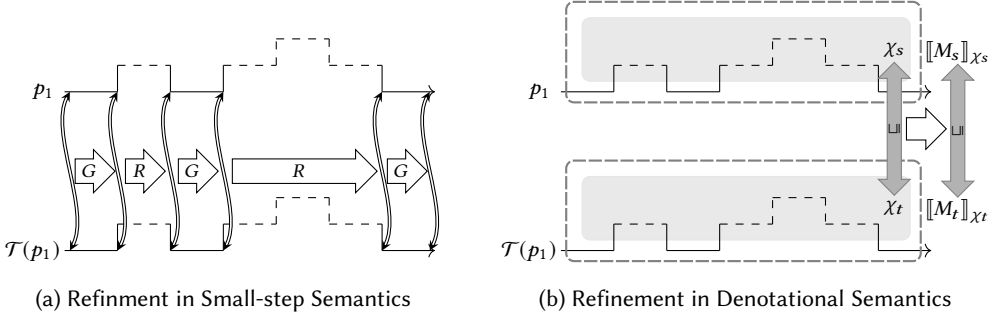


Fig. 7. Comparison between the refinement process for small-step semantics and that for denotational semantics, where  $R$  and  $G$  respectively represent rely and guarantee conditions, vertical curved arrows denote the matching relation between the source and the target program states held throughout the execution, and vertical shaded arrows denote the refinement between function denotations.

The common style of developing compositional semantics for supporting module-level compositionality is based on small-step semantics (including [Berlinger et al. \[2014\]](#), [Stewart et al. \[2015\]](#), [Song et al. \[2020\]](#), [Koenig and Shao \[2021\]](#), [Zhang et al. \[2023\]](#)), in which the description of behavior refinement must introduce extra mechanisms to capture inter-module function calls. Because C functions in a module can both be called by other modules and call other modules, a C module’s compilation correctness described by small-step semantics looks like the following, as shown in Fig. 7a:

- If an incoming call in the target language corresponds to a call in the source language (i.e., the arguments and programs states match with each other), then the first outgoing calls would match; and further, if the returned value and program states of these two first outgoing calls match with each other, then the second outgoing calls would match, etc. until the original incoming call returns.
- For horizontal compositionality, the compilation correctness of internal functions relies on external calls satisfying certain well-behavedness conditions (known as rely-conditions; e.g., external calls do not modify the private memory of callers). In turn, the execution of internal functions itself guarantees some well-behavedness conditions (known as guarantee-conditions, e.g., they do not modify the private memory of their calling environments).

In contrast, the definition of our behavior refinement is much different. Since the behavior of open modules is modeled as a function from callee’s denotation to caller’s denotation, i.e., the behavior of every external call is interpreted by semantic oracle  $\chi_s$  (for the source module) and  $\chi_t$  (for the target module), our compilation correctness is described as (shown in Fig. 7b): if the behavior of external source functions is refined by the behavior of external source functions, then the denotation of source module is refined by the denotation of the compiled target module. The refinement of semantic oracles (i.e., callee’s behavior) provides rely conditions for the refinement of caller’s behavior, and the refinement of caller’s behavior itself satisfies guarantee conditions.

[Ramananandro et al. \[2015\]](#) develop a compositional semantics on top of small-step semantics by modeling the behavior of external function calls as special events. Such a special event records the function name and the memory state before and after the external call. When composing the denotation of multiple functions, one has to interpret these special traces, in which when an external function is called, instead of asking an oracle to obtain its semantics, the special event trace of the invoked function is used to interpret it, and so on. For example, consider two procedures

$p$  and  $q$  that form two separate modules and do not use any memory state for simplicity. If the behavior of  $p$  is sequentially calling  $q$  three times, and the behavior of  $q$  is sequentially outputting zero for two times, then their semantics can be  $\llbracket p \rrbracket = \text{Extcall}(q) :: \text{Extcall}(q) :: \text{Extcall}(q) :: \text{nil}$  and  $\llbracket q \rrbracket = \text{OUT}(0) :: \text{OUT}(0) :: \text{nil}$ , where  $\text{Extcall}(q)$  is a special event and  $\text{OUT}(0)$  is the original event of CompCert. After the semantic linking, the behavior of  $p$  would be outputting zero for six times, i.e.,  $\text{OUT}(0) :: \text{OUT}(0) :: \text{OUT}(0) :: \text{OUT}(0) :: \text{OUT}(0) :: \text{OUT}(0) :: \text{nil}$ . That is, they compose semantics by replacing special events (based on the small-step semantics of CompCert), which can be viewed as progressing incrementally over the program's execution time. The key to proving behavior refinement for them is to establish the correspondence between events step by step. Therefore, their main proof structure is similar to that of small-step semantics.

Interaction tree [Xia et al. 2020] also provides a denotational approach for modeling recursive, effectual computations that can interact with their environment. Much different from the textbook denotational semantics and the denotational semantics used in this paper, interaction trees are executable via code extraction, making them suitable for debugging, testing, and implementing software artifacts. Defining refinement (a.k.a. weak simulation) in interaction trees uses coinduction (similar to the greatest fixed point used in this paper). In comparison, our formalization directly uses event trace to capture observable behaviors.

## 9 RELATED WORK

**Denotational semantics.** The foundational work of Dana Scott and Christopher Strachey [Scott 1970; Scott and Strachey 1971] in domain theory provides a mathematical framework for the denotational semantics of deterministic programs. Following their work, later efforts [Apt and Plotkin 1981; Back 1983; Broy et al. 1978; Park 1979] attempt to extend the framework for supporting non-deterministic programs, especially programs that can produce an infinite number of different results and yet be certain to terminate. Among them, Back [Back 1983] describes a path semantics for nondeterministic assignment statements indicated by semantic models described for CSP programs [Francez et al. 1979] and data flow programs [Kosinski 1978] based on paths; the relational style of denotational definition in this paper is inspired from the relational semantics proposed by Park et al. [Park 1979], which originally aims to address unbounded nondeterminism in the fair scheduling problem. This means that although each intermediate language of CompCert is deterministic, our implementation is also available for non-deterministic programs. Most recently, denotational semantics is also adapted to build the semantic model of probabilistic programming languages such as Barthe et al. [2018] and Wang et al. [2019a].

**Compositional compiler verification.** To our surprise, despite the long history of research on denotational semantics and its good compositionality, very little work has actually applied it to scenarios of realistic compiler verification. Many existing work has extended the small-step semantics of CompCert for supporting cross-module semantic linking. Among them, Ramananandro et al. [Ramananandro et al. 2015] model the behavior of external function calls as a special event recording the function name and memory states before and after function calls so as to signal state transitions made by the environment and then big-step the small-step semantics to obtain the compositional semantics for semantic linking. Beringer et al. [Beringer et al. 2014] propose a novel interaction model, called core semantics (also known as interaction semantics), that describes communication between a local thread with its environment, which is widely used by later efforts such as CompComp, i.e., Compositional CompCert [Stewart et al. 2015], CompCertM [Song et al. 2020] and CASCompCert [Jiang et al. 2019]. Specifically, with the interaction semantics, CompComp proves a general correctness result for semantic linking where programs can be

written with heterogeneous languages such as C and assembly code. However, to achieve horizontal (module-level) and vertical compositionality, it introduces complicated *structured simulations* which require a large number of changes to the original proofs of CompCert.

Compared to CompComp, SepCompCert [Kang et al. 2016] greatly reduces the complexity of proofs by limiting all source modules to be compiled by the same compiler. Furthermore, both CompCertM [Song et al. 2020] and CompCertO [Koenig and Shao 2021] have developed a lightweight verification technique for supporting semantic linking of heterogeneous languages. CompCertM proposes a RUSC (Refinement Under Self-related Contexts) theory for composing two open simulations together, while CompCertO characterizes compiled program components directly in terms of their interaction with each other and achieves compositionality through a careful and compositional treatment of calling conventions. Recently, Zhang et al. [Zhang et al. 2023] have proposed an fully composable and adequate approach to verified compilation with direct refinements between open modules based on CompCertO. We believe that their approach to handling calling conventions of heterogeneous languages can be applied to our framework.

Daniel Patterson and Amal Ahmed [Patterson and Ahmed 2019] make a comprehensive conclusion of compositional compiler correctness. They take existing compiler correctness theorems as a spectrum which ranges from CompCert, SepCompCert, CompCertX [Gu et al. 2015; Wang et al. 2019b], CompComp up to compilers verified with multi-language technique. Kumar et al. [Kumar et al. 2014] have formally verified a compiler for a functional language called CakeML, which is originally based on big-step semantics. We believe that our approach can also be used for it.

**Relationship between unified semantic operators and Kleene algebra.** A Kleene algebra (KA) [Kozen 1994] is a tuple  $(A, +, \cdot, 0, 1)$  together with a unary operator  $*$  :  $A \rightarrow A$  satisfying certain axioms (e.g., the associative law and commutative law of  $+$ ), where  $A$  is a carrier set containing identity elements  $0$  (for binary operator  $+$  :  $A \times A \rightarrow A$ ) and  $1$  (for binary operator  $\cdot$  :  $A \times A \rightarrow A$ ).

In fact, the semantic operators  $\cup, \circ, \emptyset, \mathbb{I}$  defined in §2 almost form a Kleene algebra on a semantic domain  $D$ , i.e.,  $(D, \cup, \circ, \emptyset, \mathbb{I})$  as long as we define  $R^* \triangleq R^0 \cup R \cup R \circ R \cup \dots \cup R^n$ , so that semantic equivalence can be verified in a purely equational subsystem using the axioms of Kleene algebra. Besides, the **test** operator, as an essential ingredient for modeling conventional programming constructs such as conditionals and while loops, is also widely used in extensions of Kleene algebra (known as Kleene algebra with test [Kozen and Patron 2000; Kozen and Smith 1996]).

We use Kleene fixed point to define the terminating behavior of a program instead of the asterisk form (i.e.,  $*$ ), since in addition to dealing with termination as traditional Kleene algebras do, we also need to deal with divergence, and we find that separately using Kleene fixed point and Knaster-Tarski fixed point to define termination and divergence would be better.

## 10 CONCLUSION

We have proposed a denotation-based framework for compositional compiler verification, which we conclude in the following three aspects.

- *Semantic definitions.* We extend relational semantics to realistic settings in an easy-to-formal manner, and propose unified set operators for better proof reuse, thus solving the limitations of traditional powerdomains in compiler verification scenarios. More importantly, we define a novel semantic linking operator based on fixed-point theorems, such that its equivalence with syntactic linking is reduced into concise fixed-point properties.
- *Behavior refinements.* We propose a refinement algebra to unify various forms of refinement relations, and algebraically verify compiler correctness in a unified framework.

- *Applications.* We define the denotational semantics for the front-end languages of CompCert, reprove the compilation correctness from Clight to Cminor, and support module-level compositionality in a language-independent way.

## REFERENCES

- Krzysztof R Apt and Gordon D Plotkin. 1981. A Cook's tour of countable nondeterminism. In *International Colloquium on Automata, Languages, and Programming*. Springer, 479–494.
- Ralph-Johan Back. 1983. A Continuous Semantics for Unbounded Nondeterminism. *Theor. Comput. Sci.* 23 (1983), 187–210. [https://doi.org/10.1016/0304-3975\(83\)90055-5](https://doi.org/10.1016/0304-3975(83)90055-5)
- Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 117–144. [https://doi.org/10.1007/978-3-319-89884-1\\_5](https://doi.org/10.1007/978-3-319-89884-1_5)
- Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 107–127. [https://doi.org/10.1007/978-3-642-54833-8\\_7](https://doi.org/10.1007/978-3-642-54833-8_7)
- Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 460–475. [https://doi.org/10.1007/11813040\\_31](https://doi.org/10.1007/11813040_31)
- Manfred Broy, Rupert Gnatz, and Martin Wirsing. 1978. Semantics of Nondeterministic and Noncontinuous Constructs. In *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, Germany (Lecture Notes in Computer Science, Vol. 69)*, Friedrich L. Bauer and Manfred Broy (Eds.). Springer, 553–592. <https://doi.org/10.1007/BFb0014683>
- Nissim Francez, C. A. R. Hoare, Daniel J. Lehmann, and Willem P. de Roever. 1979. Semantics of Nondeterminism, Concurrency, and Communication. *J. Comput. Syst. Sci.* 19, 3 (1979), 290–308. [https://doi.org/10.1016/0022-0000\(79\)90006-0](https://doi.org/10.1016/0022-0000(79)90006-0)
- Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. <https://doi.org/10.1145/2676726.2676975>
- Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards certified separate compilation for concurrent programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 111–125. <https://doi.org/10.1145/3314221.3314595>
- Jecheon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 178–190. <https://doi.org/10.1145/2837614.2837642>
- Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1095–1109. <https://doi.org/10.1145/3453483.3454097>
- Paul R. Kosinski. 1978. A Straightforward Denotational Semantics for Non-Determinant Data Flow Programs. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 214–221. <https://doi.org/10.1145/512760.512783>
- Dexter Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Inf. Comput.* 110, 2 (1994), 366–390. <https://doi.org/10.1006/INCO.1994.1037>
- Dexter Kozen and Maria-Christina Patron. 2000. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1861)*, John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.). Springer, 568–582. [https://doi.org/10.1007/3-540-44957-4\\_38](https://doi.org/10.1007/3-540-44957-4_38)
- Dexter Kozen and Frederick Smith. 1996. Kleene Algebra with Tests: Completeness and Decidability. In *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers (Lecture Notes in Computer Science, Vol. 1258)*, Dirk van Dalen and Marc Bezem (Eds.). Springer, 244–259. [https://doi.org/10.1007/3-540-63172-0\\_43](https://doi.org/10.1007/3-540-63172-0_43)
- Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

- POPL '14, San Diego, CA, USA, January 20-21, 2014, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. <https://doi.org/10.1145/2535838.2535841>
- Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. <https://doi.org/10.1145/1538788.1538814>
- Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. <https://doi.org/10.1007/s10817-009-9155-4>
- David Michael Ritchie Park. 1979. On the Semantics of Fair Parallelism. In *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings (Lecture Notes in Computer Science, Vol. 86)*, Dines Bjørner (Ed.). Springer, 504–526. [https://doi.org/10.1007/3-540-10007-5\\_47](https://doi.org/10.1007/3-540-10007-5_47)
- Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 85:1–85:29. <https://doi.org/10.1145/3341689>
- Gordon Plotkin. 1983. Domains. *University of Edinburgh* (1983).
- Gordon D. Plotkin. 1976. A Powerdomain Construction. *SIAM J. Comput.* 5, 3 (1976), 452–487. <https://doi.org/10.1137/0205035>
- Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, Xavier Leroy and Alwen Tiu (Eds.). ACM, 3–14. <https://doi.org/10.1145/2676724.2693167>
- Dana Scott. 1970. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford.
- Dana S Scott and Christopher Strachey. 1971. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group Oxford.
- Michael B. Smyth. 1978. Power Domains. *J. Comput. Syst. Sci.* 16, 1 (1978), 23–36. [https://doi.org/10.1016/0022-0000\(78\)90048-X](https://doi.org/10.1016/0022-0000(78)90048-X)
- Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. <https://doi.org/10.1145/3371091>
- Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 275–287. <https://doi.org/10.1145/2676726.2676985>
- Di Wang, Jan Hoffmann, and Thomas W. Reps. 2019a. A Denotational Semantics for Low-Level Probabilistic Programs with Nondeterminism. In *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2019, London, UK, June 4-7, 2019 (Electronic Notes in Theoretical Computer Science, Vol. 347)*, Barbara König (Ed.). Elsevier, 303–324. <https://doi.org/10.1016/J.ENTCS.2019.09.016>
- Yuting Wang, Pierre Wilke, and Zhong Shao. 2019b. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. <https://doi.org/10.1145/3290375>
- Glynn Winskel. 1985. On Powerdomains and Modality. *Theor. Comput. Sci.* 36 (1985), 127–137. [https://doi.org/10.1016/0304-3975\(85\)90037-4](https://doi.org/10.1016/0304-3975(85)90037-4)
- Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. <https://doi.org/10.1145/3371119>
- Ling Zhang, Yuting Wang, Jérémie Koenig, and Zhong Shao. 2023. A Bottom-Up Approach to a Unified Semantic Interface for Verified Compositional Compilation. *CoRR* abs/2302.12990 (2023). <https://doi.org/10.48550/arXiv.2302.12990> arXiv:2302.12990

## A SEMANTIC ANALYSIS FOR WHILE LOOPS

### A.1 Nondeterminism and Powerdomains

The semantics of non-deterministic programs is given by a recursively defined function mapping to a subset of domain  $D$ , where  $D$  contains interpretations of all possible running results. To apply the least fixed point theorem for continuous functions on CPOs, researchers have proposed various powerdomain constructions from the ground domain  $D$  so as to augment the system of domains used in the Scott-Strachey style of description. Consider a flat domain  $(D, \leq_D)$ , satisfying for any  $x$  and  $y$  in  $D$ ,

$$x \leq_D y \Leftrightarrow (x = \perp \vee x = y)$$

A powerdomain of  $D$ , written  $\mathcal{M}(D)$ , is a complete poset involving the subsets of the ground domain  $D$  and a new partial order  $\sqsubseteq$  on these subsets. For any  $X$  and  $Y$  in  $\mathcal{M}(D)$ , there are generally three natural ways to construct a new ordering:

$$X \sqsubseteq_0 Y \Leftrightarrow \forall x \in X, \exists y \in Y, x \leq_D y$$

$$X \sqsubseteq_1 Y \Leftrightarrow \forall y \in Y, \exists x \in X, x \leq_D y$$

$$X \sqsubseteq_2 Y \Leftrightarrow (\forall x \in X, \exists y \in Y, x \leq_D y) \wedge (\forall y \in Y, \exists x \in X, x \leq_D y)$$

**The Hoare powerdomain.** The Hoare powerdomain is established with the first ordering  $\sqsubseteq_0$  meaning that everything  $X$  can do,  $Y$  can do better. However, this ordering is just a preorder but not a partial order since it fails antisymmetry. An appropriate solution is to define an equivalence relation:  $X \sim Y$  iff  $X \sqsubseteq_0 Y$  and  $Y \sqsubseteq_0 X$ . Thus, every set  $X$  in  $\mathcal{M}(D)$  is equivalent to its *downward* closure, written  $\downarrow X$ :

$$\downarrow X \triangleq \{x_0 \in D \mid \exists x \in X, x_0 \leq_D x\}$$

In particular, a downward closed set is called a *downset*, i.e.,  $\downarrow X = X$ . The Hoare powerdomain focuses just on the downsets as being the meaningful sets on which we will reach an angelic semantics for nondeterminism, since the downsets always include element  $\perp$ . That is, in the following example, program 1 and 3 will have the same semantics.

*Example A.1.* Let `anynat()` produce a natural number nondeterministically and consider the following four programs in WHILE:

- |  |   |
|--|---|
| <ol style="list-style-type: none"> <li>1. <code>skip</code></li> <li>2. <code>while true do skip</code></li> <li>3. <code>choice (while true do skip) skip</code></li> </ol> | <ol style="list-style-type: none"> <li>4. <code>while (y == 2    x &gt; 0) do</code><br/> <code>  if (y == 2)</code><br/> <code>  then y = 1; x = anynat()</code><br/> <code>  else x = x - 1</code></li> </ol> |
|--|---|

**The Smyth powerdomain.** In contrast, the Smyth powerdomain is obtained from the preorder  $\sqsubseteq_1$  which says that everything  $Y$  can do is approximated by some behavior of  $X$ . Likewise with this ordering every set  $X$  in  $\mathcal{M}(D)$  is equivalent to its *upward* closure, written  $\uparrow X$ :

$$\uparrow X \triangleq \{x_0 \in D \mid \exists x \in X, x \leq_D x_0\}$$

The elements of the Smyth powerdomain are at least upward closed sets (*upsets*). As a result, we will achieve a demonic semantics since any program that can diverge has the semantics  $\uparrow \{\perp\} = D$ . For instance, programs 2 and 3 will have the same semantics under this powerdomain.

**The Plotkin powerdomain.** The Plotkin powerdomain uses the intersection of the first two orderings, which is also known as the Egli-Milner ordering. With this ordering, every set  $X$  in  $\mathcal{M}(D)$  is equivalent to its *convex* closure:

$$\text{conv}(X) \triangleq \{x_1 \in D \mid \exists x_0, x_2 \in X, x_0 \leq_D x_1 \leq_D x_2\}$$

The elements of the Plotkin powerdomain are sets closed under the addition of all intermediate elements. With this powerdomain all the first three programs in Example A.1 will have different meanings as one would expect. However, some constructs like program 4 that can produce infinitely many different results and yet be certain to terminate, i.e., unbounded nondeterminism, are excluded (see [Back 1983] for details). In other words, the semantics obtained from the Plotkin powerdomain is limited to settings where the execution of a program only can produce a finite number of different results, or otherwise the execution may not terminate.

As is indicated in the introduction, none of the three powerdomains is perfect and can be used to define denotational semantics for compiler verification. Park [Park 1979] pointed out that this does not at all rule out denotations obtained as subsets of domains and what seems to require careful formulation is the problem of choosing appropriate domains for use. As suggested by Park, we adapt the relational semantics and formalize it as denotations composed of multiple sets, making the algebraic ideas as accessible as possible.

## A.2 Semantics of the While Statement

Firstly, consider the denotation of termination case for the **while** statement with the function  $f$ :

$$f(x) = \text{test}(\llbracket e \rrbracket.(ffs)) \cup \text{test}(\llbracket e \rrbracket.(tts)) \circ \llbracket c \rrbracket.(nrm) \circ x$$

Then we know  $\llbracket \text{while } e \text{ do } c \rrbracket.(nrm)$  will be a fixed point of  $f$ , i.e.,

$$\llbracket \text{while } e \text{ do } c \rrbracket.(nrm) = \text{test}(\llbracket e \rrbracket.(ffs)) \cup \text{test}(\llbracket e \rrbracket.(tts)) \circ \llbracket c \rrbracket.(nrm) \circ (\llbracket \text{while } e \text{ do } c \rrbracket.(nrm))$$

It indicates that for any  $(\sigma_0, \sigma) \in \llbracket \text{while } e \text{ do } c \rrbracket.(nrm)$ , either  $\sigma_0 \in \llbracket e \rrbracket.(ffs)$  and  $\sigma = \sigma_0$ , or there exists  $\sigma_1$  such that executing the loop body one time from  $\sigma_0$  reaches  $\sigma_1$  and restarting the cycle from  $\sigma_1$  finally reaches  $\sigma$ , i.e.,  $(\sigma_1, \sigma) \in \llbracket \text{while } e \text{ do } c \rrbracket.(nrm)$ .

For any fixed point  $X$  of  $f$ , i.e.,  $X = f(X)$ , we then analyse the properties that  $X$  should have. Consider an initial state  $\sigma_0 \in \text{state}$ , and we have:

- If  $\sigma_0$  does not satisfy  $e$ , i.e.,  $\sigma_0 \in \llbracket e \rrbracket.(ffs)$ , then (1)  $(\sigma_0, \sigma_0) \in \text{test}(\llbracket e \rrbracket.(ffs))$ , (2) for any  $\sigma \neq \sigma_0$ ,  $(\sigma_0, \sigma) \notin \text{test}(\llbracket e \rrbracket.(ffs))$ , and (3) for any  $\sigma$ ,  $(\sigma_0, \sigma) \notin \text{test}(\llbracket e \rrbracket.(tts))$ . That is, if  $\sigma_0 \in \llbracket e \rrbracket.(ffs)$ , then  $\forall \sigma \in \text{state}$ ,  $(\sigma_0, \sigma) \in X$  iff  $\sigma = \sigma_0$ .
- If  $\sigma_0$  satisfies  $e$ , namely  $\sigma_0 \in \llbracket e \rrbracket.(tts)$ , and the **while** loop ends at state  $\sigma_n$  normally after executing the loop body  $n$  ( $n > 0$ ) times, then  $\forall \sigma \in \text{state}$ ,  $(\sigma_0, \sigma) \in X$  iff  $\sigma = \sigma_n$ .

The two cases indicate that in the termination case, any fixed point of  $f$  can capture the expected meanings of the while statement. If the while loop does not terminate, we expect that there will not exist any  $\sigma$  s.t.  $(\sigma_0, \sigma) \in X$ . For example, we expect the program 2 in Example A.1 to satisfy  $\llbracket \text{while true do skip} \rrbracket.(nrm) = \emptyset$ , though in this case  $f(x) = \emptyset \cup \mathbb{1} \circ \mathbb{1} \circ x = x$  and any subset of binary relations on  $\text{state}$  will be its fixed point. This indicates the  $\llbracket \text{while } e \text{ do } c \rrbracket.(nrm)$  should be defined as the least fixed point of function of  $f$ .

Secondly, considering the denotation of diverging case with the function  $g$ :

$$g(x) = \text{test}(\llbracket e \rrbracket.(tts)) \circ (\llbracket c \rrbracket.(dvg) \cup \llbracket c \rrbracket.(nrm) \circ x)$$

Obviously  $\llbracket \text{while } e \text{ do } c \rrbracket.(dvg)$  will be a fixed point of  $g$ , i.e.,

$$\llbracket \text{while } e \text{ do } c \rrbracket.(dvg) = \text{test}(\llbracket e \rrbracket.(tts)) \circ (\llbracket c \rrbracket.(dvg) \cup \llbracket c \rrbracket.(nrm) \circ \llbracket \text{while } e \text{ do } c \rrbracket.(dvg))$$

It means that for any  $\sigma_0 \in \llbracket \text{while } e \text{ do } c \rrbracket.(dvg)$ ,  $\sigma_0$  can always pass through the loop condition and then lead to divergence either by the execution of loop body or by the loop itself.

Then we analyse the properties of fixed points of  $g$ . For any  $X$  such that  $X = g(X)$ ,

- any state that does not satisfy  $e$  is not included by  $X$ , and then any state from which the execution of the loop may terminate is excluded;

- for any  $\sigma_0 \in \llbracket e \rrbracket.(\text{tts})$ , if  $\sigma_0 \in \llbracket c \rrbracket.(\text{dvg})$ , then  $\forall \sigma \in \text{state} \in X$  iff  $\sigma = \sigma_0$ , and
- for any arriving state  $\sigma_n$  by executing the loop body  $n$  times from  $\sigma_0$ , if  $\sigma_n \in \llbracket e \rrbracket.(\text{tts})$  and  $\sigma_n \in \llbracket c \rrbracket.(\text{dvg})$ , then  $\forall \sigma \in \text{state}, \sigma \in X$  iff  $\sigma = \sigma_n$ .

In summary, any fixed points of  $g$  can correctly classify the diverging behavior for the while statement. However, most of them cannot totally cover the cases where the loop body terminate normally every time and yet the loop itself are not terminate. For example, in terms of program 2,  $g(x) = 1 \circ (\emptyset \cup 1 \circ x) = x$ . Any subset of state will be a fixed point of it and what we really expect for  $\llbracket \text{while true do skip} \rrbracket.(\text{dvg})$  is the greatest one, namely the whole set state. This indicates the  $\llbracket \text{while } e \text{ do } c \rrbracket.(\text{dvg})$  should be defined as the greatest fixed point of function  $g$ .

## B SEMANTICS OF COMPCERT FRONT-END LANGUAGES

### B.1 Languages and Semantic Domain

**Syntax of Clight.** The full syntax of Clight statements  $\text{Clit.stmt}$ , ranged over by  $s$ , can be found in [Blazy et al. 2006] and we present what we are interested in as follows.

$$\begin{aligned}
 s &\triangleq \mathbf{skip} \mid s_1; s_2 \mid \mathbf{if}(e) \ s_1 \ \mathbf{else} \ s_2 \mid \mathbf{loop} \ s_1 \ s_2 \mid \mathbf{call} \ id^? \ e \ e^* \mid \\
 &\quad \mathbf{break} \mid \mathbf{continue} \mid \mathbf{return} \ e^? \mid \mathbf{switch} \ e \ (o^? : s)^* \mid \dots \\
 f &\triangleq (\text{parameter } id_i \text{ of type } \pi_i)^* : \text{return type } \pi \\
 &\quad \{ (\text{local addressable variable } id_j \text{ of type } \pi_j)^*; \\
 &\quad (\text{local non-addressable variable } id_k \text{ of type } \pi_k)^*; \\
 &\quad \text{function body } s_f \}
 \end{aligned}$$

Following Blazy et al's convention, for a syntactic construct  $x$  here we use  $x^?$  to denote the optional occurrence of  $x$  and  $x^*$  for zero or multiple times occurrence of  $x$ . All the above definitions are self-explained except for the **loop** statement which will repeatedly execute  $s_1$  and then  $s_2$ , and a “continue” jump in  $s_1$  will branch to  $s_2$  such that the definition C loops can be derived from it. For example, the **for** statement can be defined as:

$$\mathbf{for}(s_1; e; s_2) \{s_3\} \triangleq s_1; (\mathbf{loop} \ (\mathbf{if} \ (e) \ \mathbf{then} \ \mathbf{skip} \ \mathbf{else} \ \mathbf{break}); s_3 \ s_2)$$

**Semantic domain for Clight.** The program state of Clight  $\text{Clit.state}$ , as shown below, is defined as the Cartesian product of the global environment  $\text{Clit.genv}$ , the local environment  $\text{Clit.env}$ , the temporary environment  $\text{tenv}$  and the memory state  $\text{mem}$  whose definition is shared by all the intermediate languages of CompCert. We refer to [Blazy et al. 2006; Leroy 2009b] for the details of these notions.

$$\begin{aligned}
 \text{Clit.state} &\triangleq \text{Clit.genv} \times \text{Clit.env} \times \text{tenv} \times \text{mem}. \\
 \text{Clit.fstate} &\triangleq \text{Clit.genv} \times \text{mem}. \\
 \text{call\_info} \{fstate\} &\triangleq \text{id} \times \text{val}^* \times fstate.
 \end{aligned}$$

Eliminating the internal environment of a function, we obtain the set of function states  $\text{Clit.fstate}$ . Similarly we will have  $\text{Cshm.genv}$ ,  $\text{Cshm.state}$ ,  $\text{Cshm.fstate}$  for Csharpminor and  $\text{Cmin.genv}$ ,  $\text{Cmin.state}$ ,  $\text{Cmin.fstate}$  for Cminor, and the prefix name will be omitted if there is no ambiguity. The set of calling information  $\text{call\_info}$  is the Cartesian product of the set of function identifiers  $\text{id}$ , the set of arguments  $\text{val}^*$  and the given function states set  $fstate$ , where  $\text{val}$  is the set of values ranging over 32-bit integers, 64-bit floats, memory locations, and an undefined value that represents for instance the value of uninitialized variables. Then the semantic domain for Clight statements is shown as follows.

$$\text{Record Clit.Denote: Type} := \{$$

```

nrm: state → event* → state → Prop;
brk: state → event* → state → Prop;
ctn: state → event* → state → Prop;
rtn: state → event* → state → val → Prop;
err: state → event* → Prop;
c11: state → event* → call_info → Prop;
fin_dvg: state → event* → Prop;
inf_dvg: state → event∞ → Prop
}.

```

Firstly, CompCert divides program behavior into four categories: terminating, aborting, silently diverging, and reacting behavior. In our case, the first two are denoted by the sets `nrm` and `err` respectively, and the latter two, to make it clearer, are called finite divergence and infinite divergence, denoted by the sets `fin_dvg` and `inf_dvg` respectively. Secondly the sets `brk`, `ctn`, `rtn` are used to manipulate the control flow of Clight featured with **break**, **continue**, and **return** statements. This means for any initial state  $\sigma_0$ , the execution of a fragment of Clight programs from  $\sigma_0$  will eventually reach state  $\sigma_1$  and then exit because of the **break**, **continue**, or **return** statement (with a return value), producing a sequence of input-output events. Finally, the set `c11` denotes that executing the program from some  $\sigma_0$  will reach a program point that generates a function call recorded as an element of `call_info`.

**Syntax of Csharpminor and Cminor.** Csharpminor is an untyped low-level imperative language featured with infinite loops, blocks and early block exits, and its syntax is shown as follows.

$$\begin{aligned}
u &\triangleq \mathbf{skip} \mid u_1; u_2 \mid \mathbf{if}(e) \ u_1 \ \mathbf{else} \ u_2 \mid \mathbf{loop} \ \{u\} \mid id^? = e(e^*) : sig \mid \\
&\quad \mathbf{block}\{u\} \mid \mathbf{exit}(n) \mid \mathbf{return} \ e^? \mid \mathbf{switch} \ b \ e \ (o^? : u)^* \mid \dots \\
f &\triangleq (\text{parameter } id_i)^* : \text{signature } sig \\
&\quad \{ \text{(local addressable variable } id_j \text{ of size } z_j)^* ; \\
&\quad \quad \text{(local non-addressable variable } id_k)^* ; \\
&\quad \text{function body } u_f \}
\end{aligned}$$

The syntax of Cminor is almost the same as that of Csharpminor. In addition to further converting the switch statement into a simpler jump table form, the main feature of this language is the pre-allocation of stack space for local addressable variables, as is shown below.

$$\begin{aligned}
t &\triangleq \mathbf{skip} \mid t_1; t_2 \mid \mathbf{if}(e) \ t_1 \ \mathbf{else} \ t_2 \mid \mathbf{loop} \ \{t\} \mid id^? = e(e^*) : sig \mid \\
&\quad \mathbf{block}\{t\} \mid \mathbf{exit}(n) \mid \mathbf{return} \ e^? \mid \mathbf{switch} \ b \ e \ tbl \ n \mid \dots \\
f &\triangleq (\text{parameter } id_i)^* : \text{signature } sig \\
&\quad \{ \text{stack size } z; \text{(local non-addressable variable } id_k)^* ; \\
&\quad \text{function body } u_f \}
\end{aligned}$$

**Semantic domain for Csharpminor and Cminor.** To this end, the Csharpminor and Cminor enjoys the same form of denotation in which only their state sets differ.

```

Cshm.state  $\triangleq$  Cshm.genv  $\times$  env  $\times$  tenv  $\times$  mem.
Cmin.state  $\triangleq$  Cmin.genv  $\times$  val  $\times$  tenv  $\times$  mem.
Record Cshm.Denote: Type := {
  nrm: state → event* → state → Prop;
  blk: nat → state → event* → state → Prop; (* NEW *)
  rtn: state → event* → state → val → Prop;

```

```

err: state  $\rightarrow$  event*  $\rightarrow$  Prop;
c11: state  $\rightarrow$  event*  $\rightarrow$  call_info  $\rightarrow$  Prop;
fin_dvg: state  $\rightarrow$  event*  $\rightarrow$  Prop;
inf_dvg: state  $\rightarrow$  event $^\infty$   $\rightarrow$  Prop
}.

```

Compared to Clight, the local environment of Csharpminor<sub>env</sub> maps a local variable to its memory locations only while that of Clight maps a local variable to its block identifier and the data type of it. For Cminor, since all local variables whose addresses are taken are stored on the stack, its program states record the stack pointer of type `val` but no longer the local environment. In addition, the control flow of them is structured with the **block** and **exit** statement. For instance, the execution of statement `block { block {  $s_1$ ; exit(1);  $s_2$ ;  $s_3$  }` is equivalent to the execution of  $s_1; s_3$  since the statement **exit**( $n$ ) terminates prematurely the execution of the  $(n+1)$  layers of nested **block** statements. This behavior is captured by the set `blk`, namely for any  $(n, \sigma_0, \tau, \sigma_1) \in \text{blk}$ , executing the program from  $\sigma_0$  will reach state  $\sigma_1$  and  $n$  records the layers of nested blocks to exit. The meaning of other fields are the same as that of Clight.

```

Record FDenote {F V A B C T}: Type := {
  dom: F  $\rightarrow$  Prop;
  nrm: T  $\rightarrow$  F  $\rightarrow$  V*  $\rightarrow$  A  $\rightarrow$  B*  $\rightarrow$  A  $\rightarrow$  V  $\rightarrow$  Prop;
  err: T  $\rightarrow$  F  $\rightarrow$  V*  $\rightarrow$  A  $\rightarrow$  B*  $\rightarrow$  Prop;
  c11: T  $\rightarrow$  F  $\rightarrow$  V*  $\rightarrow$  A  $\rightarrow$  B*  $\rightarrow$  C  $\rightarrow$  Prop;
  fin_dvg: T  $\rightarrow$  F  $\rightarrow$  V*  $\rightarrow$  A  $\rightarrow$  B*  $\rightarrow$  Prop;
  inf_dvg: T  $\rightarrow$  F  $\rightarrow$  V*  $\rightarrow$  A  $\rightarrow$  B $^\infty$   $\rightarrow$  Prop
}.

```

The denotation of functions are polymorphically defined by parameterizing the type of function names `F`, the type of variable values `V`, the type of states `A`, the type of events `B`, the type of calling information `C` and the type of normally terminating behavior of callees `T`. Thus, function denotation for different languages can be instantiated from the `FDenote`. For instance, the denotation of Clight functions is instantiated as:

```

TN { fstate }  $\triangleq$  id  $\times$  val  $\times$  fstate  $\times$  event*  $\times$  fstate  $\times$  val.
Clit.FDenote  $\triangleq$  @FDenote id val Clit.fstate event (@TN Clit.fstate)).

```

Given the normally terminating behavior of callees, a function name in the set of valid names (i.e., `dom`) and its arguments, the function's execution may either terminate with a return value (if not, an undefined value will be returned), abort, finitely or infinitely diverge, and these behaviors are captured by sets `nrm`, `err`, `fin_dvg` and `inf_dvg` of `FDenote` respectively.

## B.2 Semantics of Clight

Traditionally, the semantic function for Clight statements  $C : \text{stmt} \rightarrow \text{TN} \rightarrow \text{Denote}$  is parameterized by callee's behavior `TN`, written as  $\llbracket s \rrbracket_\chi$  for given statement  $s \in \text{stmt}$  and behavior  $\chi \subseteq \text{TN}$ . Let  $\mathbb{1} \triangleq \{(\sigma, \text{nil}, \sigma) \mid \sigma \in \text{state}\}$  and then we selectively list some key cases for the definition of  $C$ . Among them, the cases of if and sequential statements are similar to those in the WHILE language, and here we focus on the semantics of control flow, loops and function calls.

$\llbracket \text{skip} \rrbracket_{\chi, (\text{nrm})} \triangleq \mathbb{1}$ , and other fields are assigned the empty set.  
 $\llbracket \text{break} \rrbracket_{\chi, (\text{brk})} \triangleq \mathbb{1}$ , and other fields are assigned the empty set.  
 $\llbracket \text{continue} \rrbracket_{\chi, (\text{ctn})} \triangleq \mathbb{1}$ , and other fields are assigned the empty set.  
 $N_1 = \llbracket s_1 \rrbracket_{\chi, (\text{nrm})} \cup \llbracket s_1 \rrbracket_{\chi, (\text{ctn})} \quad N_2 = N_1 \circ \llbracket s_2 \rrbracket_{\chi, (\text{nrm})}$   
 $\llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi, (\text{brk})} = \llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi, (\text{ctn})} \triangleq \emptyset$

$$\begin{aligned} \llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}.(\text{brk}) &\triangleq \mu x. \llbracket s_1 \rrbracket_{\chi}.(\text{brk}) \cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{brk}) \cup N_2 \circ x \\ \llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}.(\text{rtn}) &\triangleq \mu x. \llbracket s_1 \rrbracket_{\chi}.(\text{rtn}) \cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{rtn}) \cup N_2 \circ x \end{aligned}$$

Firstly, for the denotation of the break (continue) statement, only the brk (ctn) field is a reflexive relation, and other fields are assigned the empty set, which means that once a break (continue) statement is encountered, the execution will end prematurely. It can be imaged that the denotation of the return statement is defined in a similar way, except for coping with the return value additionally.

Next we discuss the denotational semantics of Clight loops. For convenience, let  $N_1$  denote that the loop body  $s_1$  either ends normally or prematurely due to a continue statement in  $s_1$  and  $N_2$  denote a “sequential” execution of the loop bodies  $s_1$  and  $s_2$ . Since the loop itself will not induce the statements following it to end prematurely, the brk and ctn fields of its denotation are both assigned the empty set. The execution of loops will normally terminate because of a break statement in either  $s_1$  or  $s_2$  after executing the loop body several times. In the same way, we can define the denotation of exiting cases by a return statement.

The interesting cases are those for finitely diverging and infinitely diverging behavior of Clight loops. We use silent and non-silent operators to explicitly filter silent and non-silent event traces when defining the diverging behavior of Clight loops, as shown below: given sets  $A, B$ , and relations  $X \subseteq A \times B^* \times A$ ,  $Y \subseteq A \times B^*$ ,

$$\begin{aligned} \blacktriangle X &\triangleq \{(\sigma, \tau, \sigma') \mid (\sigma, \tau, \sigma') \in X \wedge \tau \neq \text{nil}\} \\ \triangle X &\triangleq \{(\sigma, \tau, \sigma') \mid (\sigma, \tau, \sigma') \in X \wedge \tau = \text{nil}\} \\ \Delta Y &\triangleq \{(\sigma, \tau) \mid (\sigma, \tau) \in Y \wedge \tau = \text{nil}\} \end{aligned}$$

Then the diverging behavior of Clight loops is defined as follows.

$$\begin{aligned} \llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}.(\text{fin\_dvg}) &\triangleq \mu x. \llbracket s_1 \rrbracket_{\chi}.(\text{fin\_dvg}) \cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{fin\_dvg}) \\ &\quad \cup \Delta \mathbb{K} \cup N_2 \circ x \\ \text{where } \mathbb{K} &= \nu x. \llbracket s_1 \rrbracket_{\chi}.(\text{fin\_dvg}) \cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{fin\_dvg}) \cup N_2 \circ x \\ \llbracket \text{loop } s_1 \ s_2 \rrbracket_{\chi}.(\text{inf\_dvg}) &\triangleq \nu x. \mathbb{B} \circ \mathbb{D} \cup (\mathbb{B} \circ \blacktriangle N_2) \circ x \\ \text{where } \mathbb{B} &= \mu x. \mathbb{1} \cup \Delta N_2 \circ x, \text{ and} \\ \mathbb{D} &= \llbracket s_1 \rrbracket_{\chi}.(\text{inf\_dvg}) \cup N_1 \circ \llbracket s_2 \rrbracket_{\chi}.(\text{inf\_dvg}) \end{aligned}$$

The former indicates that the loop, after being executed several times, will diverge since the loop body  $s_1$  or  $s_2$  does, or the entire loop itself silently diverges. Let  $\mathbb{B}$  denote the behavior of silently executing the loop body a finite number of times, and  $\mathbb{D}$  denote the non-silently diverging behavior due to the execution of loop body  $s_1$  or  $s_2$ . Then, the later indicates that either the loop body diverges non-silently after executing it a finite number of times, or a non-silent event occurs after executing the loop body a finite number of times and then this process repeats infinitely. As we will see soon, defining the diverging behavior of recursive function calls suffers from the same problem as that of loops when evaluating a module’s semantics in §B.4, and it is solved in the same way.

$$\begin{aligned} \llbracket \text{call } id^? \ e \ e^* \rrbracket_{\chi}.(\text{brk}) &\triangleq \{((g_e, l_e, t_e, m), \tau, (g_e, l_e, t'_e, m')) \mid \exists h \ v^* \ r, \\ &\quad (h, v^*) \in \text{eval\_args}(e, e^*, (g_e, l_e, t_e, m)) \wedge \\ &\quad (h, v^*, (g_e, m), \tau, (g_e, m'), r) \in \chi \wedge \\ &\quad t'_e = \text{set\_temp}(id^?, r, t_e) \} \end{aligned}$$

$$\llbracket \text{call } id^? e e^* \rrbracket_{\chi.(c11)} \triangleq \{((g_e, l_e, t_e, m), \text{nil}, (h, v^*, \theta)) \mid \theta = (g_e, m) \wedge (h, v^*) \in \text{eval\_args}(e, e^*, (g_e, l_e, t_e, m))\}$$

Last but not least, the normal termination of a call statement defined above means that starting from the program state  $\sigma = (g_e, l_e, t_e, m)$ , we first evaluate the function name and parameters of the callee according to  $e$  and  $e^*$  respectively. Then by the denotation of callees  $\chi$ , we obtain the function state  $(g_e, m')$  and the return value  $r$  after the function call. Finally, the temporary environment  $t_e$  is updated for local variable  $id$  (if any) with return value  $r$ . At the same time, its `c11` field records the calling information, i.e., the callee's function name, parameters and current function state. Other fields except `err` are assigned the empty set.

The semantic function for Clight functions  $\mathcal{F} : \text{id} \times \text{function} \rightarrow \text{FDenote}$ , written as  $\llbracket (h, f) \rrbracket$  for a given function name  $h$  and its definition  $f$ , are defined as follows.

$$\begin{aligned} \llbracket (h, f) \rrbracket_{\chi.(dom)} &\triangleq \{h\}, \text{ i.e., the singleton set including function name } h \text{ only} \\ \llbracket (h, f) \rrbracket_{\chi.(nrm)} &\triangleq \{(h, v^*, (g_e, m_0), \tau, (g_e, m_3), r) \mid \exists l_e t_e m_1, \\ &\quad \text{function\_entry}(g_e, f, v^*, m_0, l_e, t_e, m_1) \wedge \\ &\quad \exists \sigma \sigma' m_2, \sigma = (g_e, l_e, t_e, m_1) \wedge \sigma' = (g_e, \_, \_, m_2) \wedge \\ &\quad (((\sigma, \tau, \sigma') \in \llbracket s_f \rrbracket_{\chi.(nrm)} \wedge r = \text{vundef}) \vee (\sigma, \tau, \sigma', r) \in \\ &\quad \llbracket s_f \rrbracket_{\chi.(rtn)}) \wedge \text{free\_list}(m_2, \text{block\_of\_env}(g_e, l_e)) = [m_3]\} \\ \llbracket (h, f) \rrbracket_{\chi.(c11)} &\triangleq \{(h, v^*, (g_e, m_0), \tau, \delta) \mid \exists l_e t_e m_1, \\ &\quad \text{function\_entry}(g_e, f, v^*, m_0, l_e, t_e, m_1) \wedge \\ &\quad \exists \sigma, \sigma = (g_e, l_e, t_e, m_1) \wedge (\sigma, \tau, \delta) \in \llbracket s_f \rrbracket_{\chi.(c11)}\} \end{aligned}$$

The terminating behavior of a function is recorded by the `nrm` set and is denoted according to  $f$ 's function body  $s_f$  which would exit normally or exit by a **return** statement. Specifically, the local environment  $l_e$ , temporary environment  $t_e$  and memory state after initialization of the local variables  $m_1$  are evaluated from the function definition  $f$  with its parameters  $v^*$  and the initial function state  $(g_e, m_0)$ . Then after executing the function body  $s_f$ , the return value  $r$ , and memory state  $m_2$  are known through the denotational semantics of  $s_f$ . Finally the memory state  $m_3$  is updated from  $m_2$  by releasing the local variables of  $f$ . Other fields like `c11` are defined similarly, i.e., the behavior of the function is determined by the behavior of its body.

### B.3 Semantics of Csharpminor and Cminor

For the denotational semantics of Csharpminor and Cminor, most of the content is similar to Clight, and the difference lies in the manipulation of control flow shown as follows.

$$\begin{aligned} \llbracket \text{block}\{u\} \rrbracket_{\chi.(nrm)} &\triangleq \llbracket u \rrbracket_{\chi.(nrm)} \cup \llbracket u \rrbracket_{\chi.(blk)}_0 \\ \llbracket \text{block}\{u\} \rrbracket_{\chi.(blk)} &\triangleq \{(n, \sigma, \tau, \sigma') \mid (n+1, \sigma, \tau, \sigma') \in \llbracket u \rrbracket_{\chi.(blk)}\} \\ \llbracket \text{exit}(n) \rrbracket_{\chi.(blk)} &\triangleq \{(n_0, \sigma, \text{nil}, \sigma) \mid n_0 = n\}, \text{ other fields are assigned } \emptyset \end{aligned}$$

where  $(\text{blk})_n \triangleq \{(\sigma, \tau, \sigma') \mid (n, \sigma, \tau, \sigma') \in \text{blk}\}$  for any  $n \in \text{nat}$ . The above definitions mean that the block statement will normally terminate if its internal statement  $u$  terminates normally or exits one layer of block execution prematurely; and it will early exit  $n$  layers if  $u$  early exits  $n+1$  layers of block execution. The other fields of  $\llbracket \text{block}\{u\} \rrbracket_{\chi.(blk)}$  are directly determined by the corresponding set of the internal statement  $u$ . On the other hand, when an exit statement is encountered, the following statements will no longer execute normally and the layer of blocks to be exited early are recorded by the set `blk`.

### B.4 Semantics of Modules

Before defining the semantics of a module, we first need an auxiliary function  $\mathcal{F}^* : (\text{id} \times \text{function})^* \rightarrow \text{FDenote}$ , written as  $\llbracket M \rrbracket$ , which maps a list of functions to the union of their denotations<sup>7</sup>, i.e.,

$$\llbracket (h_1, f_1); \dots; (h_n, f_n) \rrbracket \triangleq \llbracket (h_1, f_1) \rrbracket \cup \dots \cup \llbracket (h_n, f_n) \rrbracket$$

We then discuss the definition of semantic function  $\mathcal{M} : (\text{id} \times \text{function})^* \rightarrow \text{FDenote}$  for a module  $M$ , written as  $\llbracket M \rrbracket$ , which maps a list of functions within the module to the “full” denotation of each function.

First of all, given the denotation of callees outside the module  $\chi : \mathbb{T}\mathbb{N}$ , the normally terminating behavior of each function in module  $M$ , namely  $\llbracket M \rrbracket_{\chi}.(\text{nrn})$ , is defined as follows:

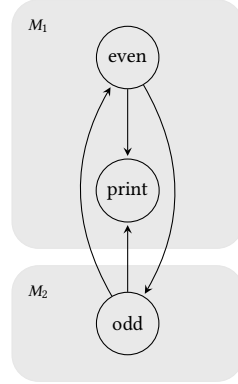
$$\llbracket M \rrbracket_{\chi}.(\text{nrn}) \triangleq \mu \chi_0. \llbracket M \rrbracket_{\chi_0 \cup \chi}.(\text{nrn})$$

Recall that the least fixed point of  $f(\chi_0) = \llbracket M \rrbracket_{\chi_0 \cup \chi}.(\text{nrn})$  is  $\sqcup \{f^i(\emptyset) \mid i = 0, 1, \dots\}$ . The intuition behind it is that with more and more times of iteration, we can constantly approximate the semantics (similar to the definition of loops’ semantics) that a function can normally terminate after a finite number of calls.

```

1  bool even(int n) {
2    print(n);
3    if (n == 0) return true;
4    else return odd(n - 1);
5  }
6  void print(int n) {
7    printf("%d ", n);
8  }
9  bool odd(int n) {
10   print(n);
11   if (n == 0) return false;
12   else return even(n - 1);
13 }
```

(a) The Parity Judgement Program



(b) Calling Relation Between Them

Fig. 8. Check if a Number  $n$  is Odd or Even

For example, consider a parity judgement program implemented by two mutually recursive functions shown in Fig. 8a. The even function tells us that a number  $n$  is even if  $n$  is 0, or  $(n - 1)$  is odd; the odd function says that  $n$  is odd if  $(n - 1)$  is even. Both of them they call a print function for displaying the value of  $n$ . Their calling relation and residing modules are shown in Fig. 8b.

Let an event of outputting an number  $n$  be  $\text{output}(n)$ . For the first iteration  $\chi_0 = \emptyset$ , and then  $\llbracket M_1 \rrbracket_{\chi}.(\text{nrn})(\text{even}) = \emptyset$ ,  $\llbracket M_1 \rrbracket_{\chi}.(\text{nrn})(\text{print}) = \{(\text{print}, n, \theta, \tau, \theta, \text{Vundef}) \mid \theta \in \text{fstate} \wedge \tau = \text{output}(n)\}$ .

For the second iteration,  $\llbracket M_1 \rrbracket_{\chi}.(\text{nrn})(\text{even}) = \{(\text{even}, 0, \theta, \tau, \theta, \text{true}) \mid \theta \in \text{fstate} \wedge \tau = \text{output}(0)\} \cup \{(\text{even}, n, \theta, \tau, \theta', r) \mid n > 0 \wedge \exists \tau', (\text{odd}, n-1, \theta, \tau', \theta', r) \in \chi \wedge \tau = \text{output}(n) \cdot \tau'\}$  and  $\llbracket M_1 \rrbracket_{\chi}.(\text{nrn})(\text{print})$  is unchanged. In this way, we can obtain the terminating behavior of a module correctly after a finite number of iterations.

<sup>7</sup>where the union of two elements in domain FDenote is defined as the union of their corresponding sets.

For a given  $\chi : \mathbb{T}\mathbb{N}$ , let  $\chi_n = \llbracket M \rrbracket_{\chi} \cdot (\text{nrm}) \cup \chi$ . The other cases for a module's denotation semantics are defined as follows<sup>8</sup>.

$$\begin{aligned}
\odot(M) &\triangleq \{(\delta, \text{nil}, \delta) \mid \exists f v^* \theta, \delta = (f, v^*, \theta) \wedge f \notin \langle M \rangle \cdot (\text{dom})\} \\
\llbracket M \rrbracket_{\chi} \cdot (\text{c11}) &\triangleq \mu\chi_0. \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \odot(M) \cup \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
\llbracket M \rrbracket_{\chi} \cdot (\text{err}) &\triangleq \mu\chi_0. \langle M \rangle_{\chi_n} \cdot (\text{err}) \cup \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
\llbracket M \rrbracket_{\chi} \cdot (\text{fin\_dvg}) &\triangleq \mu\chi_0. \langle M \rangle_{\chi_n} \cdot (\text{fin\_dvg}) \cup \Delta \mathbb{K}(M) \cup \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
&\quad \text{where } \mathbb{K}(M) \triangleq \nu\chi_0. \langle M \rangle_{\chi_n} \cdot (\text{fin\_dvg}) \cup \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
\llbracket M \rrbracket_{\chi} \cdot (\text{inf\_dvg}) &\triangleq \nu\chi_0. \mathbb{B} \circ \langle M \rangle_{\chi_n} \cdot (\text{inf\_dvg}) \cup (\mathbb{B} \circ \blacktriangle \langle M \rangle_{\chi_n} \cdot (\text{c11})) \circ \chi_0 \\
&\quad \text{where } \mathbb{B} = \mu\chi_0. \mathbb{1} \cup \Delta \langle M \rangle_{\chi_n} \cdot (\text{c11}) \circ \chi_0
\end{aligned}$$

where  $\odot(M)$  defines an identity relation on the set of external calls.

The  $\llbracket M \rrbracket_{\chi} \cdot (\text{c11})$  means that a function inside the module  $M$  will eventually call a function outside the module after a finite number of internal calls. For example, the calling denotation of even in module  $M_1$  is defined as

$$\llbracket M_1 \rrbracket_{\chi} \cdot (\text{c11})(\text{even}) = \{(\text{even}, n, \theta, \tau, \delta) \mid \delta = (\text{odd}, n-1, \theta) \wedge \tau = \text{output}(n)\}.$$

Similarly,  $\llbracket M \rrbracket_{\chi} \cdot (\text{err})$  represents aborting behavior inside the module, i.e., a function inside the module errors if its internal statements abort or the function passes through a finite number of function calls and one of them fails. Furthermore, as we define the semantics of the loop statement, we follow the same way to distinguish silent and nonsilent divergence of function calls. That is, a function either diverges from the function body of a certain callee, or it gets stuck in infinitely many recursive calls.

## B.5 Semantic Linking

Following the approach to the evaluation of a module's semantics, we can easily define the semantic linking between modules, written as  $\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket$ . Firstly, given the denotation of callees outside the two module  $M_1$  and  $M_2$ , namely  $\chi : \mathbb{T}\mathbb{N}$ , the normally terminating behavior of each function within them  $\llbracket M \rrbracket_{\chi} \cdot (\text{nrm})$ , is defined as follows:

$$(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{nrm}) \triangleq \mu\chi_0. (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_0 \cup \chi} \cdot (\text{nrm})$$

It indicates that with the behavior of functions outside the two modules, we can reach the semantics of a given function by a finite iteration of function calls. Analogy to evaluating the semantics of each function in a module (shown in §B.4), the semantics of each function within the two modules can be evaluated by treating their denotations as if they reside in one module. Once again, for a given  $\chi : \mathbb{T}\mathbb{N}$ , let  $\chi_n = (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{nrm}) \cup \chi$ . Then the other fields of  $\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket$  are defined as follows.

$$\begin{aligned}
(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{err}) &\triangleq \mu\chi_0. (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{err}) \cup \\
&\quad (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{c11}) &\triangleq \mu\chi_0. (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \odot(M_1 + M_2) \cup \\
&\quad (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{fin\_dvg}) &\triangleq \mu\chi_0. (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{fin\_dvg}) \cup \Delta \mathbb{K} \cup \\
&\quad (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \chi_0
\end{aligned}$$

<sup>8</sup>Let the currying and uncurrying between  $\text{id} \times \text{val}^* \times \text{fstate} \times \text{event}^* \times \text{call\_info}$  and  $\text{call\_info} \times \text{event}^* \times \text{call\_info}$  happen as needed so that we freely use the sequential composition operator.

$$\begin{aligned}
& \text{where } \mathbb{K} = \nu\chi_0.(\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{fin\_dvg}) \cup (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \chi_0 \\
& (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{inf\_dvg}) \triangleq \nu\chi_0. \mathbb{B} \circ (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{inf\_dvg}) \cup \\
& \quad (\mathbb{B} \circ \blacktriangle (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11})) \circ \chi_0 \\
& \text{where } \mathbb{B} = \mu\chi_0. \mathbb{1} \cup \Delta (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_n} \cdot (\text{c11}) \circ \chi_0
\end{aligned}$$

With the semantic linking defined by taking fixed points, we are able to demonstrate the equivalence between semantic linking and syntactic linking in an algebraic style. In this process, it is necessary to show some coinciding properties of fixed points, such as Lemma B.1 and Lemma B.2.

LEMMA B.1 (COINCIDE THEOREM 1). *Given a CPO  $(A, \subseteq)$ , for any monotonic and continuous functions  $f : A \rightarrow A$  and  $g : A \rightarrow A$ ,*

$$\mu z. (\mu x. f(x \cup z) \cup \mu x. g(x \cup z)) \equiv \mu x. (f(x) \cup g(x))$$

PROOF. Let  $L(z) = \mu x. f(x \cup z) \cup \mu x. g(x \cup z)$ , and  $R(x) = f(x) \cup g(x)$ . We have to prove  $\mu L \subseteq \mu R$  and  $\mu R \subseteq \mu L$ . Here we show the proof of the former only, since the latter is even easier to prove.

$\mu L \subseteq \mu R$ : It is sufficient to show that  $\forall n, L^n(\emptyset) \subseteq \mu R$ . By taking induction on  $n$ , the key is to prove: if  $L^n(\emptyset) \subseteq \mu R$ , then  $L(L^n(\emptyset)) \subseteq \mu R$ , i.e.,  $\mu x. f(x \cup L^n(\emptyset)) \cup \mu x. g(x \cup L^n(\emptyset)) \subseteq \mu R$ .

Let  $F(x) = f(x \cup L^n(\emptyset))$ , and  $G(x) = g(x \cup L^n(\emptyset))$ . It is sufficient to show that  $\mu F \subseteq \mu R$  and  $\mu G \subseteq \mu R$ . These two cases can be proved similarly, and we show the proof for the first case: It's sufficient to prove  $\forall m, F^m(\emptyset) \subseteq \mu R$ . By taking induction on  $m$ , the key is to prove if  $F^m(\emptyset) \subseteq \mu R$ , then  $F(F^m(\emptyset)) \subseteq \mu R$ , which holds since  $f(F^m(\emptyset) \cup L^n(\emptyset)) \subseteq f(\mu R \cup \mu R) \subseteq f(\mu R) \cup g(\mu R) \subseteq \mu R$ .  $\square$

LEMMA B.2 (COINCIDE THEOREM 2). *Given sets  $A$  and  $B$ . For any relations  $E_1, E_2 \subseteq A \times B^*$  and  $N_1, N_2, M_1, M_2 \subseteq A \times B^* \times A$ , let  $f_1(x) = E_1 \cup N_1 \circ x$ ,  $f_2(x) = E_2 \cup N_2 \circ x$ ,  $g_1(x) = N_1 \circ M_1 \cup N_1 \circ x$  and  $g_2(x) = N_2 \circ M_2 \cup N_2 \circ x$ , then*

$$\mu z. (\mu f_1 \cup \mu f_2 \cup (\mu g_1 \cup \mu g_2) \circ z) \equiv \mu x. E_1 \cup E_2 \cup (N_1 \cup N_2) \circ x$$

with side condition that  $M_i \circ N_i \equiv \emptyset$ ,  $M_i \circ E_i \equiv \emptyset$ ,  $M_i \circ N_j \equiv N_j$ ,  $M_i \circ E_j \equiv E_j$  for  $i = 1, 2$  and  $i \neq j$ .

The Lemma B.2 says that  $(N_1^+ \circ M_1 \cup N_2^+ \circ M_2)^* \circ (N_1^* \circ E_1 \cup N_2^* \circ E_2) \equiv (N_1 \cup N_2)^* \circ (E_1 \cup E_2)$  where  $X^+$  denotes the transitive closure of  $X$  and  $X^*$  denotes the reflexive-transitive closure of  $X$  for a relation  $X \subseteq A \times B^* \times A$ .

THEOREM B.3 (EQUIVALENCE BETWEEN SEMANTIC AND SYNTACTIC LINKING). *For any modules  $M_1$  and  $M_2$ ,  $\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket \equiv \llbracket M_1 + M_2 \rrbracket$ , where equivalence between elements of  $\text{FDenote}$  is defined as the equivalence of their corresponding sets.*

PROOF. The first case is to show that for any given  $\chi$ ,

$$\begin{aligned}
& (\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{norm}) \equiv \llbracket M_1 + M_2 \rrbracket_{\chi} \cdot (\text{norm}) \quad \mathbf{iff} \\
& \mu\chi_0. (\llbracket M_1 \rrbracket \cup \llbracket M_2 \rrbracket)_{\chi_0 \cup \chi} \cdot (\text{norm}) \equiv \mu\chi_0. \llbracket M_1 + M_2 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm}) \quad \mathbf{iff} \\
& \mu\chi_0. \llbracket M_1 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm}) \cup \llbracket M_2 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm}) \equiv \\
& \mu\chi_0. (\llbracket M_1 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm}) \cup \llbracket M_2 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm}))
\end{aligned}$$

Let  $f(\chi_0) = \llbracket M_1 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm})$  and  $g(\chi_0) = \llbracket M_2 \rrbracket_{\chi_0 \cup \chi} \cdot (\text{norm})$ . It's not hard to see  $f$  and  $g$  are monotonic and continuous, and then it's proved by applying Lemma B.1.

The second case is to show that for any given  $\chi$ ,  $(\llbracket M_1 \rrbracket \oplus \llbracket M_2 \rrbracket)_{\chi} \cdot (\text{err}) \equiv \llbracket M_1 + M_2 \rrbracket_{\chi} \cdot (\text{err})$  which can be proved by applying Lemma B.2. Other cases are proved similar to the second case with proper coincide theorems.  $\square$

## C FORMAL DEFINITION OF GAMMA INSTANCES

As we can see, there should be various instances of the gamma function for relating each field of the statement or function denotations. We next discuss how they are formally defined. Given sets  $A_1, A_2$  and  $W$ , a Kripke relation  $R : W \rightarrow \{X \mid X \subseteq A_1 \times A_2\}$  is a family of relations indexed by a Kripke world  $W$ , written as  $\mathcal{K}_W(A_1, A_2)$ . For any two sequences of events  $\tau_0$  and  $\tau_1$ , we say  $\tau_0 \leq_T \tau$  if and only if  $\tau_0$  is the prefix of  $\tau$ . Let  $(W, \leq_W)$  be a preorder. For given sets  $N_s$  and  $E_s$ , we define instances of the gamma function  $\gamma_{\text{nm}}, \gamma_{\text{rtn}}, \gamma_{\text{c11}}$  and  $\gamma_{\text{dvg}}$  as follows:

$$\begin{aligned}
\forall (\sigma_t, \tau, \sigma'_t) \in \gamma_{\text{nm}}(N_s, E_s) \text{ iff } & \forall w \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow \\
& \exists w' \sigma'_s, (\sigma_s, \tau, \sigma'_s) \in N_s \wedge w \leq_W w' \wedge (\sigma'_s, \sigma'_t) \in R(w') \\
& \vee \exists \tau_0, (\sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau. \\
\forall (\sigma_t, \tau, \sigma'_t, r) \in \gamma_{\text{rtn}}(N_s, E_s) \text{ iff } & \forall w \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow \\
& \exists w' \sigma'_s r', (\sigma_s, \tau, \sigma'_s, r') \in N_s \wedge w \leq_W w' \wedge \\
& (r, r') \in V(w') \wedge (\sigma'_s, \sigma'_t) \in R(w') \\
& \vee \exists \tau_0, (\sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau. \\
\forall (\sigma_t, \tau, \delta_t) \in \gamma_{\text{c11}}(N_s, E_s) \text{ iff } & \forall w \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow \\
& \exists w' \delta_s, (\sigma_s, \tau, \delta_s) \in N_s \wedge w \leq_W w' \wedge (\delta_s, \delta_t) \in \Delta(w') \\
& \vee \exists \tau_0, (\sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau. \\
\forall (\sigma_t, \tau) \in \gamma_{\text{dvg}}(N_s, E_s) \text{ iff } & \forall w \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow \\
& (\sigma_s, \tau) \in N_s \vee \exists \tau_0, (\sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau.
\end{aligned}$$

Note that each gamma instance may be parameterized with different Kripke relations for relating the states, arguments or return values as needed between the source and the target. The above definitions can be naturally extended to supporting relating function denotations. For instance, the normally terminating case for the refinement of function denotations is defined as:

$$\begin{aligned}
\forall (h, v_s^*, \sigma_t, \tau, \sigma'_t, r) \in \gamma_{\text{nm}}(N_s, E_s) \text{ iff } & \\
& \forall v_s^* w \sigma_s, (\sigma_s, \sigma_t) \in R(w) \Rightarrow (v_s^*, v_t^*) \in V^*(w) \Rightarrow \\
& \exists w' \sigma'_s r', (h, v_s^*, \sigma_s, \tau, \sigma'_s, r') \in N_s \wedge w \leq_W w' \wedge \\
& (r, r') \in V(w') \wedge (\sigma'_s, \sigma'_t) \in R(w') \\
& \vee \exists \tau_0, (h, v_s^*, \sigma_s, \tau_0) \in E_s \wedge \tau_0 \leq_T \tau.
\end{aligned}$$

## D FULL DEFINITION OF REFINEMENT BETWEEN DENOTATIONS

*Definition D.1 (Statement refinement between Clight and Csharpminor).* Given natural numbers  $n_b, n_c$  and the aborting behavior of external functions  $\chi_e \subseteq \text{call\_info} \times \text{event}^*$ . A denotation  $D_2$  of Cshml.Denote is said to be a refinement of a denotation  $D_1$  of Clit.Denote, written as  $D_2 \lesssim_{(n_b, n_c, \chi_e)} D_1$  if and only if:

$$D_2.(\text{nm}) \subseteq \gamma(D_1.(\text{nm}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (1)$$

$$D_2.(\text{blk})_{n_b} \subseteq \gamma(D_1.(\text{brk}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (2)$$

$$D_2.(\text{blk})_{n_c} \subseteq \gamma(D_1.(\text{ctn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (3)$$

$$D_2.(\text{rtn}) \subseteq \gamma(D_1.(\text{rtn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (4)$$

$$D_2.(\text{err}) \subseteq \gamma(\emptyset, D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (5)$$

$$D_2.(\text{c11}) \subseteq \gamma(D_1.(\text{c11}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (6)$$

$$D_2.(\text{fin\_dvg}) \subseteq \gamma(D_1.(\text{fin\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (7)$$

$$D_2.(\text{inf\_dvg}) \subseteq \gamma(D_1.(\text{inf\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (8)$$

**Definition D.2 (Statement refinement between Csharpminor and Cminor).** Given the exit environment  $\xi$ , and the aborting behavior of external functions  $\chi_e \subseteq \text{call\_info} \times \text{event}^*$ . A denotation  $D_2$  of  $\text{Cmin.Denote}$  is said to be a refinement of a denotation  $D_1$  of  $\text{Cshm.Denote}$ , written as  $D_2 \lesssim_{(\xi, \chi_e)} D_1$  if and only if condition (1) and (4) ~ (8) in Def. D.1 hold, and additionally:

$$\forall n, D_2.(\text{blk})_{\text{shift}(\xi, n)} \subseteq \gamma(D_1.(\text{blk})_n, D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \quad (9)$$

The definition of refinement between function denotations is similar to the definition of refinement between statement denotations.

**Definition D.3 (Refinement between function denotation).** For any elements  $F_1$  and  $F_2$  in  $\text{FDenote}$ ,  $F_2$  is said to be a refinement of  $F_1$ , written as  $F_2 \sqsubseteq F_1$  if and only if for any  $\chi_t, \chi_n$  and  $\chi_e$  such that  $\chi_t \subseteq \gamma(\chi_n, \chi_e)$ , the following conditions hold, where  $D_1 = (F_1)_{\chi_s}$  and  $D_2 = (F_2)_{\chi_t}$ :

$$\begin{aligned} D_2.(\text{dom}) &\equiv D_1.(\text{dom}) \\ D_2.(\text{nrn}) &\subseteq \gamma(D_1.(\text{nrn}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{err}) &\subseteq \gamma(\emptyset, D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{c11}) &\subseteq \gamma(D_1.(\text{c11}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{fin\_dvg}) &\subseteq \gamma(D_1.(\text{fin\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \\ D_2.(\text{inf\_dvg}) &\subseteq \gamma(D_1.(\text{inf\_dvg}), D_1.(\text{err}) \cup D_1.(\text{c11}) \circ \chi_e) \end{aligned}$$

## E COQ COMPARISON WITH COMPCERT AND COMPCERTOU

We name our work as VComp, and name [Zhang et al. 2023] as CompCertOU, which is based on CompCertO. Generally speaking, compared with small-step semantics, we differ with them in

- The definition of states;
  - CompCert/CompCertOU: There are three constructors of state for small-step semantics: `ItnlState`, `CallState`, `RtrnState`. For example, the state of `Clight`:

```
state = | ItnlState: function → statement → cont → env → temp_env → mem → state
        | Callstate: val → list val → cont → mem → state
        | RtrnState: val → cont → mem → state.
```

- VComp: `state = temp_env × mem`.
- We do not have to relate *continuations* in matching due to denotational definition.
- Initialization of function entry and memory release of function exit. The location of their semantic definition is different.
  - CompCert/CompCertOU: at the definition of `step`, following the semantics of function calls;
  - VComp: at the definition of functions' denotation.

More specifically, a pass-to-pass comparison is shown as follows.

**Clight to Csharpminor.** All the constraints are the same except for:

- CompCert/CompCertOU: `match_env` resides in `match_states`;
- VComp: `[Cshmgenproof] match_env` is a side precondition for statement refinement, which is satisfied at the moment of function entry.

### Csharpminor to Cminor.

- For CompCertOU:

- $\text{injp} = \text{world} \times (\text{world} \rightarrow \text{meminj}) \times \text{rel world} \times R_{\text{world}}(\text{mem}, \text{mem})$ , where  $\text{injp\_acc } w \ w' \in \text{rel world}$  is defined as (informally):
  - + *ro\_unchanged*: readonly memory is unchanged;
  - + *max\_perm\_decrease*: **The maximum permission is decremented**;
  - + *unchanged\_on (loc\_out\_of\_reach j sm<sub>1</sub>)*: **outside mapped locations by j from sm<sub>1</sub> is unchanged**;
  - + *unchanged\_on (unmapped j)*: **unmapped location is unchanged**;
  - + *inject\_incr*: **memory injection is increased**;
  - + *inject\_separated*: **except newly increased, original injection is unchanged**.

These constraints make  $\text{injp\_acc}$  a preorder, which may not be kept if removing some of them.

- *match\_states* constrains:

- + *SPFresh*: spwan new stack pointer from the old one;
- + *MINJ*:  $\text{Mem.inj } f \text{ sm } \text{tm}$ ; **memory injection**;
- + *MCS*: **match callstack**;
  - \* *MTemp*: **temporary environment is related**;
  - \* *MEnv*: **relating local environment to stack pointer**;
  - \* *SPBoundt*: **stack pointer is within nextblock of memory**
  - \* *MBounds*: the addresses of variables in loc-env with permissions are within variable size;
  - \* *PaddingFreeable*, used for releasing the memory of local variables when exiting function;

- Comparison with VComp:

- Only the following (state-related) constraints remain in *match\_states*:
  - + *MINJ*:  $\text{Mem.inj } f \text{ sm } \text{tm}$
  - + *MTemp*: **temporary environment is related**;
- Other constraints are moved to:
  - + well-definedness of semantics: *nextblock\_incr*, maybe part of  $\text{injp\_acc}$  such as unchanged properties on the unmapped or out-of-reach memory.
  - + side precondition for statement refinement: *MEnv*, *MEnv*, *SPBoundt*, *HiBounds*, *HiBoundt*.
- Remark that
  - + *SPFresh* is removed since calling stack is not involved in our semantics;
  - + *MBounds* and *PaddingFreeable* is removed from *match\_states*. They are established in function entry, and preserved since the  $\text{injp\_acc}$  of statement refinement;
  - + *match\_env* is less constrained, eliminating low bounds and high bounds. CompCert and CompCertOU originally require the addresses of local variables are within  $(\text{lo}, \text{hi})$ ;
  - + CompCertOU use self-defined (*match\_genvs*) to build relationship between global environments. We keep the *match\_globenv* in CompCert, so we additionally state the high bound of global definitions.