# VeriFast: Sound Symbolic Linking in the Presence of Preprocessing

*Gijs Vanspauwen*
*Bart Jacobs*

*Report CW 638, June 2013*

# VeriFast: Sound Symbolic Linking in the Presence of Preprocessing

*Gijs Vanspauwen*
*Bart Jacobs*

*Report CW 638, June 2013*

Department of Computer Science, KU Leuven

**Abstract**

Formal verification enables developers to provide safety and security guarantees about their code. A modular verification approach supports the verification of different pieces of an application in separation. VeriFast is an annotation-based verifier for C source code that implements symbolic linking to support modular verification. This report describes the process of symbolic linking and the unsoundness introduced by the C preprocessor. Moreover it contains a detailed formalization of our solution and a proof of its correctness.

# VeriFast: Sound Symbolic Linking in the Presence of Preprocessing

Gijs Vanspauwen    Bart Jacobs

June 10, 2013

## Contents

## 1 Introduction

VeriFast[1] is a tool that allows developers of C programs to prove certain properties of their code. To help VeriFast prove these properties, some hints in the form of annotations must be added to the code and these annotations are just comments as far as the C compiler is concerned.

Working with the C programming language almost always means working with the C preprocessor. The preprocessor provides the facilities of header file inclusion, of macro definition, of macro expansion and of conditional compilation to the developer. While this functionality is very useful for the developer, the preprocessor introduces some difficulties when combining it with VeriFast. The main problem is that soundness of symbolic linking is not guaranteed.

Symbolic linking is a process for deciding whether or not earlier verified source files can be safely linked together (i.e. earlier proven properties remain valid). The problem with combining it with preprocessing is that the result of a header expansion depends upon the defined macros at that point. So a header file may have a different meaning for the different source files that include it. A technique that ensures soundness of symbolic linking in the presence of preprocessing, is presented in this text.

As far as the C preprocessor is concerned, a source file consists of words and preprocessor directives. This fact is reflected in the definition of a token in Section 2. Such a token represents a C source file and will serve as input to the formalized preprocessor. Before formalizing the preprocessor in Section 4, some auxiliary definitions are given in Section 3. In Section 5 the concept symbolic linking is explained. Then, the symbolic link soundness problem that manifests itself when using the preprocessor is explained in Section 6. In Section 7, a permissive preprocessing technique is formally described under which symbolic linking remains sound, this technique is based upon the concept of context-free preprocessing. Finally, in Section 8 a proof of soundness is presented and in Section 9, the implementation of this technique in VeriFast is described.

## 2 Grammar for Tokens

A token represents the contents of a C source file as seen by the C preprocessor. Assume that the following is given:

- W: set of words
- $w \in$ W: a word
- $\bar{w} \in$ W$^*$: a list of words
- H: set of header names
- $h \in$ H: a header name

Then the BNF in Definition 1 defines the set of tokens T.

[1]VeriFast, http://people.cs.kuleuven.be/~bart.jacobs/verifast/

**Definition 1.**
$t$ ::=
    $\bar{w}$
    $t\ t$
    $h$
    ***def*** $w\ \bar{w}$
    ***undef*** $w$
    ***ifdef*** $w\ t$ ***else*** $t$ ***endif***

A token can be just a list of words, a sequence of other tokens or a preprocessor directive. There are four possible preprocessor directives in this simplified setting:

- header inclusion - $h$
- macro definition - **def** $w\ \bar{w}$
- macro removal - **undef** $w$
- conditional compilation - **ifdef** $w\ t$ **else** $t$ **endif**

A C source file is thus represented by a single token which will serve as input to the formalized preprocessor.

## 3 Auxiliary Definitions

Some auxiliary definitions that are needed in the rest of this text are given here. First, some notational conventions concerning functions, lists and multisets are specified. Secondly, a token $t$, a header name $h$ and a word $w$ are as in Section 2. Then the concept of defined macros is introduced. Defined macros are represented by a partial function from W to W*. The fact that macros cannot have arguments, does not limit the essential capabilities of the formalized preprocessor:

> *Every invocation of a macro with arguments can be replaced by the invocation of a new macro without arguments where the original arguments are substituted in the body.*

The set of header maps is described next. A header map is a function that associates a token with each possible header name. It will be used by the formalized preprocessor when it needs to expand a header. Then preprocessor trees are defined. A preprocessor represents the output of the preprocessor, while keeping the include structure of the input token intact. Finally declaration blocks, verified declaration blocks and verification environments are specified. These concepts are necessary to describe the process of recursive type checking which is required by symbolic linking (see Section 5).

For any set S:

- $\varnothing : \varnothing \to \mathrm{S}$:
  the empty function over S.
- For any set D:
  - the singleton function update of an f: S $\to$ D is defined as:
  $$f[s := d] = \lambda s'. \begin{cases} d & \text{if } s' = s \\ f(s') & \text{otherwise} \end{cases}$$
  - the function update of an f: S $\to$ D by a g: S $\to$ D is defined as:
  $$f \cup g = \lambda s'. \begin{cases} g(s') & \text{if } s' \in \mathit{dom}(g) \\ f(s') & \text{otherwise} \end{cases}$$
  - the domain restriction of a function p: S $\to$ D for a subset S$'$ of S:
  $$p|_{\mathrm{S}'} = \lambda s'. \begin{cases} p(s') & \text{if } s' \in \mathrm{S}' \\ \mathit{undefined} & \text{otherwise} \end{cases}$$
  - a subfunction $p'$ of a function $p$: S $\to$ D is defined as:
  $$p' \subseteq p \Leftrightarrow \begin{cases} \mathit{dom}(p') \subseteq \mathit{dom}(p) \wedge \\ \forall s.\ s \in \mathit{dom}(p') \Rightarrow p'(s) = p(s) \end{cases}$$

For any set S:

- $\bar{s} \in \mathrm{S}^*$:
  a list of elements of $S$
- $[]_S \in \mathrm{S}^*$:
  the empty list of elements of $S$ (the subscript $_S$ can be dropped when its meaning is clear from context)
- $[s_1, s_2, s_3, \ldots] \in \mathrm{S}^*$:
  another list of elements of $S$
- $[s_{11}, s_{12}, \ldots][s_{21}, s_{22}, \ldots] = [s_{11}, s_{12}, \ldots, s_{21}, s_{22}, \ldots]$:
  appending lists
- $\bar{s} = s :: \bar{s}_{rest}$:
  $s$ is the first element and $\bar{s}_{rest}$ is the tail of the non-empty list $\bar{s}$

For any set S:

- $\{|s_1, s_2, s_2, s_3|\} \in \mathbb{N}_0^S$:
  a multiset over $S$
- $\{|s_1, s_2, s_2|\} \uplus \{|s_3, s_3|\} = \{|s_1, s_2, s_2, s_3, s_3|\}$:
  union of two multisets over $S$

- $t \in \mathrm{T}$: a token
- $h \in \mathrm{H}$: a header name
- $w \in \mathrm{W}$: a word

---

- D: set of partial functions from W to $\mathrm{W}^*$
- $\mathrm{D} = \mathrm{W} \rightharpoonup \mathrm{W}^*$
- $d$: defined macros
- $d \in \mathrm{D}$

---

- $\mathrm{M}_H$: set of header maps
- $\mathrm{M}_H = \mathrm{H} \to \mathrm{T}$
- $m \in \mathrm{M}_H$: a header map
- $m \in \mathrm{H} \to \mathrm{T}$

---

- $\tau \in \mathrm{P}_{trees}$:
  a preprocessor tree, the output of the formalized preprocessor
- $\tau ::=$

  $[]$

  $w :: \tau$

  $(h, \tau) :: \tau$
- Note that by definition a preprocessor tree is a (nested) list, this fact will be used in definitions that follow.

---

- DB: set of declaration blocks
- $\mathrm{DB} = \mathrm{W}^*$
- $b \in \mathrm{DB}$:
  a declaration block

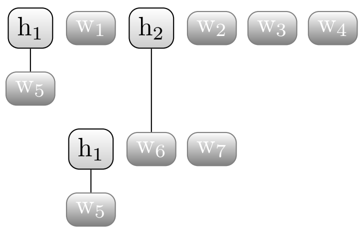

Figure 1: A graphical representation of a preprocessor tree:
$[(h_1, [w_5]), w_1, (h_2, [(h_1, [w_5]), w_6, w_7]), w_2, w_3, w_4]$

- $\mathrm{DB}_{tc}$: set of type checked declaration blocks
- $b_{tc} \in \mathrm{DB}_{tc}$:
  a type checked declaration block
- E: set of type checking environments
- $e \in \mathrm{E}$:
  a type checking environment
- $\mathrm{DB}_{tc}$ and E are the smallest sets for which:
  - $\mathrm{DB}_{tc} = \mathrm{DB} \times \mathrm{E}$
  - $\mathrm{E} \in \mathbb{N}_0^{\mathrm{DB}_{tc}}$ (multiset over $\mathrm{DB}_{tc}$)

## 4 Definition of Preprocessing

Given a certain header map $m$, the formal preprocessor will accept a token $t$ and a set of defined macros $d$ and it returns a resulting preprocessor tree $\tau$ and an updated set of defined macros $d'$. This is captured by the judgment $m \vdash (d, t) \Downarrow (d', \tau)$. The inference rules from Definition 2 describe how such a judgment can be derived. These rules describe the execution of the preprocessor using big-step semantics. The judgment actually defines a function from a specific $m$, $d$ and $t$ to a resulting $d'$ and $\tau$ as specified in Lemma 1 .

Part of the resulting output when using the inference rules from Definition 2, is a preprocessor tree. Figure 1 shows such a tree. The actual C preprocessor just outputs a list of words (called lexical tokens). To transform a preprocessor tree into a list of words, one can collapse the tree by traversing the tree depth-first and left-to-right (ignoring the header nodes). For the tree in Figure 1, this list will be:

$$[w_5,\ w_1,\ w_5,\ w_6,\ w_7,\ w_2,\ w_3\ w_4]$$

**Definition 2.**

$$\frac{}{m \vdash (d, []) \Downarrow (d, [])}\ \textsc{P-words-empty}$$

$$\frac{w \notin dom(d) \qquad m \vdash (d, \bar{w}) \Downarrow (d, \tau)}{m \vdash (d, w :: \bar{w}) \Downarrow (d, w :: \tau)}\ \textsc{P-words-undefined}$$

$$\frac{\begin{array}{c} w \in dom(d) \qquad d(w) = \bar{w}_1 \\ m \vdash (d|_{dom(d) \setminus \{w\}}, \bar{w}_1) \Downarrow (d|_{dom(d) \setminus \{w\}}, \tau_1) \qquad m \vdash (d, \bar{w}_2) \Downarrow (d, \tau_2) \end{array}}{m \vdash (d, w :: \bar{w}_2) \Downarrow (d, \tau_1\ \tau_2)}\ \textsc{P-words-defined}$$

$$\frac{m \vdash (d, t_1) \Downarrow (d', \tau_1) \qquad m \vdash (d', t_2) \Downarrow (d'', \tau_2)}{m \vdash (d, t_1 t_2) \Downarrow (d'', \tau_1\ \tau_2)}\ \textsc{P-sequence}$$

$$\frac{m \vdash (d, m(h)) \Downarrow (d', \tau)}{m \vdash (d, h) \Downarrow (d', [(h, \tau)])}\ \textsc{P-header-exp}$$

$$\frac{}{m \vdash (d, \mathbf{def}\ w\ \bar{w}) \Downarrow (d[w := \bar{w}], [])}\ \textsc{P-define}$$

$$\frac{}{m \vdash (d, \mathbf{undef}\ w) \Downarrow (d|_{dom(d) \setminus \{w\}}, [])}\ \textsc{P-undefine}$$

$$\frac{w \in dom(d) \qquad m \vdash (d, t_1) \Downarrow (d', \tau)}{m \vdash (d, \mathbf{ifdef}\ w\ t_1\ \mathbf{else}\ t_2\ \mathbf{endif}) \Downarrow (d', \tau)}\ \textsc{P-branch-if}$$

$$\frac{w \notin dom(d) \qquad m \vdash (d, t_2) \Downarrow (d', \tau)}{m \vdash (d, \mathbf{ifdef}\ w\ t_1\ \mathbf{else}\ t_2\ \mathbf{endif}) \Downarrow (d', \tau)}\ \textsc{P-branch-else}$$

**Lemma 1.**
$\forall\, m, d, d'_1, d'_2, t, \tau_{p_1}, \tau_{p_1}.$

$$(m \vdash (d, t) \Downarrow (d'_1, \tau_{p_1}) \wedge m \vdash (d, t) \Downarrow (d'_2, \tau_{p_2})) \Rightarrow$$
$$d'_1 = d'_2 \wedge \tau_{p_1} = \tau_{p_2}$$

*Proof.*
This can straightforwardly be proven by induction on the derivation $m \vdash (d, t) \Downarrow (d'_1, \tau_{p_1})$ and seeing that each possible last rule used in that derivation uniquely determines the last rule used in the derivation for $m \vdash (d, t) \Downarrow (d'_2, \tau_{p_2})$ (i.e. the same rule). Since each rule is deterministic for a certain $m$, $d$ and $t$ (by using the induction hypothesis in all inductive rules and the fact that $m$ is a function in rule P-HEADER-EXP), we know $d'_1 = d'_2$ and $\tau_{p_1} = \tau_{p_2}$. □

## 5 Symbolic Linking

VeriFast is a tool that can check certain properties (e.g. memory safety) of C programs. Verification by VeriFast requires that each function is annotated with a contract (i.e. a precondition and a postcondition) written in separation logic and each function is verified separately by applying a technique called symbolic execution. The verification of a single function is based upon the contracts of all the functions that are called inside the body of that function. To support modular verification (e.g. verification of a single source file in isolation), symbolic linking is implemented in VeriFast. However, symbolic linking can as well be added to most other annotation-based verifiers (possibly for another language than C). Adding symbolic linking to a certain verifier, will affect the verification process in a few places. These will be now discussed in turn.

## 5.1 Input constraint

The input to the verifier must adhere to the following condition: when in one source file a declaration is used that is implemented in another file (as is the case in any real world application), both files must include the same interface file containing a forward declaration (including necessary verifier annotations) of that declaration. For C programs this means that two C source files must include the same header file containing a forward declaration of the construct implemented in one source file and used in the other. If this constraint is not fulfilled then the symbolic linking process (see further) will fail.

## 5.2 Verification process

While verifying a source file and an unimplemented declaration from an interface is used, the corresponding annotations are considered as valid. It is this assumption that allows the verification of a single part of an application in isolation, i.e. modular verification.

## 5.3 Manifest files

During the verification of a source file, the verifier must generate a manifest file. A manifest file describes the provided (i.e. implemented) and required (i.e. used but not implemented) declarations in a source file. This description of a declaration contains not only the name of the declaration, but also the file where it is declared for the first time (e.g. an included header file that contains its forward declaration). The verifier can then check later that different verified source files can be safely linked together by looking at these manifest files - it is this process that is called symbolic linking.

## 5.4 Symbolic linking process

During symbolic linking, the corresponding manifest files of the input source files are compared: if for every required function (description) of some input file an implementation exists in some other input file, the verifier can conclude that it is safe to link the input files together. Note that this process does not require to reverify the source files, looking at the manifest files is sufficient.

## 5.5 Recursive context-free type checking

An important issue was ignored when describing manifest files and the symbolic linking process. Interface files can contain auxiliary constructs (inductive data types, pure functions over these data types, predicates, . . . ) for specifying annotations. If an interface uses such auxiliary constructs from another interface in its annotations, it must be made sure that the first interface file includes the second. Otherwise the semantics of an inclusion depends upon the context in which the interface was included and thus its meaning can be different for different source files that included it. A way to ensure this requirement, is to type check each (directly or indirectly) included interface recursively in isolation. If an included interface is type checked correctly in isolation, we know it includes all the necessary constructs for the semantics of its contents.

To formalize this recursive context-free type checking process, we need the concepts of a declaration block $b \in \mathrm{DB}$ (i.e. a list or words), a type checked declaration block $b_{tc} \in \mathrm{DB}_{tc}$ (i.e. a declaration block together with its type checking environment) and a type checking environment $e \in \mathrm{E}$ (i.e. a multiset of type checked declaration blocks). We will first look at the situation without recursive type checking before specifying the recursive process. Then we will present a way to compare both processes.

### 5.5.1 Type checking formalized

Let the function CP from Definition 3 represent the normal type checking procedure in the compilation process. The input to CP is a preprocessor tree $\tau$ and a current type checking environment $e$, and the output is the resulting type checking environment. This resulting type checking environment is a multiset containing all the declaration blocks found in the preprocessor tree together with the environment in which they are type checked. For a one-pass compile language like C every declaration is type checked given all the previous encountered declarations. So the rule for $\mathrm{CP}(b :: \tau, e)$ in Definition 3 correctly includes the previous current environment as the type checking environment of the encountered declaration block $b$. Note that $b$ is used here to implicitly indicate the longest match of consecutive words in the tree $\tau$ that is not interrupted by an include. The rule for $\mathrm{CP}((h, \tau_h) :: \tau, e)$ ensures that the expansion of an included header is type checked with all the previous declarations in the type checking environment. The resulting type checking environment is then used as the new current one while type checking the rest of $\tau$.

**Definition 3.**

$$\begin{array}{rcl} \mathit{CP}([], e) & = & e \\ \mathit{CP}(b :: \tau, e) & = & \mathit{CP}(\tau, e \uplus \{|(b, e)|\}) \\ \mathit{CP}((h, \tau_h) :: \tau, e) & = & \mathit{CP}(\tau, \mathit{CP}(\tau_h, e)) \end{array}$$

For a given preprocessor tree $\tau$ generated form a specific source file, the result of $\mathrm{CP}(\tau, \varnothing)$ thus represents the semantics of that source file as seen by the C compiler.

### 5.5.2 Recursive type checking formalized

As mentioned before, to support symbolic linking the type checking procedure must become recursive and the result of this recursive procedure must be presented to the verification process. In Definition 4 the function VF is defined, which represents this recursive type checking. The output of VF is, as for CP, again a type checking environment and represents the semantics of the corresponding source file as seen by the verification process. From the rule for VF$((h, \tau_h) :: \tau, e)$ it is clear that a header is type checked in isolation and the resulting declarations are added to the current type checking environment[2].

**Definition 4.**

$$\begin{array}{rcl} \mathit{VF}([], e) & = & e \\ \mathit{VF}(b :: \tau, e) & = & \mathit{VF}(\tau, e \uplus \{|(b, e)|\}) \\ \mathit{VF}((h, \tau_h) :: \tau, e) & = & \textbf{let } e' := \mathit{VF}(\tau_h, \varnothing) \textbf{ in} \\ & & \mathit{VF}(\tau, e \uplus e') \end{array}$$

### 5.5.3 Equivalence of type checking environments

If we can prove that for the preprocessor tree $\tau$ generated from a specific source file, the semantics of $CP(\tau, \varnothing)$ are the same as that of $VF(\tau, \varnothing)$, we now that (if type checking succeeds) the recursive type checking procedure has the same semantics as normal type checking So we need a way to compare type checking environments.

Comparing type checking environments can be done using the two mutually recursive judgments from Definition 5 and Definition 6. In Definition 5 the (asymmetric) judgment $\gtrapprox$ means equivalence between two type checking environments. Clearly two empty environments are equivalent. If two environments are equivalent, adding a type checked declaration block to each of them where the type checking environment of the first subsumes the one of the second as defined in Definition 6, preserves this equivalence.

**Definition 5.**

$$\frac{}{\varnothing \gtrapprox \varnothing} \textsc{Env-eq-empty}$$

$$\frac{e_1 \gtrapprox e_2 \qquad e_{11} \succeq e_{21}}{e_1 \uplus \{|(b, e_{11})|\} \gtrapprox e_2 \uplus \{|(b, e_{21})|\}} \textsc{Env-eq-not-empty}$$

**Definition 6.**

$$\forall\, e_1, e_2.\ (e_1 \succeq e_2 \Leftrightarrow \exists e_3 . e_1 \gtrapprox e_2 \uplus e_3)$$

To see why the judgment from Definition 5 indeed implies that equivalent type checking environments have the same semantics according to the C language, note that the C language has the property in Axiom 1.

**Axiom 1.** *If a declaration block can be type checked correctly with two different sets of visible declaration blocks and if one of the sets of visible blocks is a subset of the other set, then the declaration block has the same semantics relative to the two sets of visible blocks.*

[2]If header guards are used to prevent multiple includes of the same file, this technique will most likely fail. If a guarded header is included for the second time (i.e. nothing is included at all), the declarations in the header will not be in the type checking environment of the other header file that included the first. We come back to this in Section 7.

### 5.5.4 Proof of equivalence

We can now try to prove that for any preprocessor tree $\tau$ the type checking environment calculated by CP is equivalent to that calculated by VF. This ensures that the semantics of recursive type checking are the same as those of normal type checking. Theorem 1 states this more formally. Of course this can only be true if both environments can be type checked correctly. The situation in which this is not true is ignored here. But if it does occur, the implementation as described in Section 9 must signal an error. To prove Theorem 1, we first prove Lemma 2 by induction on $\tau$.

**Lemma 2.**

$$\forall\, e_1, e_2, e_3, \tau.\;\; e_1 \gtrapprox e_2 \uplus e_3 \Rightarrow [\mathit{CP}(\tau, e_1) \gtrapprox \mathit{VF}(\tau, e_2) \uplus e_3]$$

*Proof.*

The only base case for this induction:

- $\underline{\tau = []}$
  For all $e$, CP$([], e)$ is equal to $e$ and also VF$([], e)$ is equal to $e$. So if $e_1 \gtrapprox e_2 \uplus e_3$, then CP$([], e_1) \gtrapprox$ VF$([], e_2) \uplus e_3$ follows immediately.

There are two inductive cases:

- $\underline{\tau = b :: \tau_r}$
  For all $e$:

  - CP$(b :: \tau_r, e) =$ CP$(\tau_r, e \uplus \{|(b, e)|\})$
  - VF$(b :: \tau_r, e) =$ VF$(\tau_r, e \uplus \{|(b, e)|\})$

  If we know $e_1 \gtrapprox e_2 \uplus e_3$, we know that $e_1 \succeq e_2$ according to Definition 6. Then, we can deduce $e_1 \uplus \{|(b, e_1)|\} \gtrapprox e_2 \uplus \{|(b, e_2)|\} \uplus e_3$ from Definition 5. So we know by induction hypothesis that:

  - CP$(\tau_r, e_1 \uplus \{|(b, e_1)|\}) \gtrapprox$ VF$(\tau_r, e_2 \uplus \{|(b, e_2)|\}) \uplus e_3$

  Rewriting this with previous equalities gives the necessary environment equivalence:

  - CP$(b :: \tau, e_1) \gtrapprox$ VF$(b :: \tau, e_2) \uplus e_3$

- $\underline{\tau = (h, \tau_h) :: \tau_r}$
  For all $e$:

  - CP$((h, \tau_h) :: \tau_r, e) =$ CP$(\tau_r,$ CP$(\tau_h, e))$
  - VF$((h, \tau_h) :: \tau_r, e) =$ VF$(\tau_r, e \uplus$ VF$(\tau_h, \varnothing))$

  If we know $e_1 \gtrapprox e_2 \uplus e_3$, we know that $e_1 \gtrapprox \varnothing \uplus [e_2 \uplus e_3]$ and by induction hypothesis:

  - CP$(\tau_h, e_1) \gtrapprox$ VF$(\tau_h, \varnothing) \uplus [e_2 \uplus e_3]$

  which we can rewrite as:

- $\mathrm{CP}(\tau_h, e_1) \gtrapprox [e_2 \uplus \mathrm{VF}(\tau_h, \varnothing)] \uplus e_3$

Again by induction hypothesis the following holds:

- $\mathrm{CP}(\tau_r, \mathrm{CP}(\tau_h, e_1)) \gtrapprox \mathrm{VF}(\tau_r, e_2 \uplus \mathrm{VF}(\tau_h, \varnothing)) \uplus e_3$

which we can rewrite as the necessary environment equivalence:

- $\mathrm{CP}((h, \tau_h) :: \tau_r, e_1) \gtrapprox \mathrm{VF}((h, \tau_h) :: \tau_r, e_2) \uplus e_3$

□

**Theorem 1.**

$$\forall\, \tau.\ CP(\tau, \varnothing) \gtrapprox VF(\tau, \varnothing)$$

*Proof.* We can prove Theorem 1 by choosing $e_1$, $e_2$ and $e_3$ as $\varnothing$ in Lemma 2 This gives us exactly Theorem 1. □

## 6 Symbolic Linking and Preprocessing

There is a problem with the approach presented in Section 5 when the complete functionality of the C preprocessor is allowed. While the preprocessor defined in Section 4 is a simplified version of the actual C preprocessor described in the C11-standard[3], it captures the essence of the problem. The problem is illustrated by the following example:

Let the tokens $t_1$ and $t_2$ be as follows:

$t_1 = [w_1]\ h\ [w_4, w_5]$
$t_2 = [w_1]\ \mathbf{def}\ w_2\ [w_3]\ h\ [w_4, w_5]$

And let H = $\{h\}$ and:

$m(h) = [w_a, w_2, w_b]$

Then,

$m \vdash (\varnothing, t_1) \Downarrow (\varnothing, \tau_1)$
$m \vdash (\varnothing, t_2) \Downarrow (\varnothing[w_2 := [w_3]], \tau_2)$

and

$\tau_1 = [w_1, (h, [w_a,\ w_2,\ w_b]), w_4, w_5]$
$\tau_2 = [w_1, (h, [w_a,\ w_3,\ w_b]), w_4, w_5]$

The words in the output of the preprocessor for input $t_1$ that are the result of expanding the header $h$ (using inference rule P-HEADER-EXP), are $w_a\, w_2\, w_b$.

[3]C11 or formally ISO/IEC 9899:2011,
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

For $t_2$, these are $w_a w_3$, $w_b$. The result of the expansion of a header thus depends upon the set of defined macros right before the expansion takes place. The C preprocessor also has this property.

If this kind of preprocessing is used by VeriFast before the verification of C source code, it could lead to wrong conclusions during symbolic linking. To see this, consider the following scenario:

- The header file `a.h`, shown in Listing 1, forward declares the function `foo()`. The precondition contains the preprocessor symbol `BAR`, and the body of that macro determines what is required by this function to fulfill the postcondition. The postcondition, on the other hand, states that this function does not go wrong (which means among other things no null dereferences).

Listing 1: Forward declaration of function `foo()` in the header file `a.h`

```
void foo()
//@ requires BAR;
//@ ensures true;
```

- The source file `a.c`, shown in Listing 2, implements the function `foo()`. Before this function is implemented, the preprocessor symbol `BAR` is defined as `false` and the header file `a.h` is included. By defining `BAR` as `false`, the contract of the forward declaration of function `foo()` included from `a.h` corresponds to the contract of the implementation in `a.c`. The precondition of this contract now requires `false` and thus makes the contract hold trivially. That is the reason that VeriFast will approve the function `foo()` during verification, although there is clearly a memory access violation in its implementation. Verification would not succeed of course if the function was used somewhere in the file.

Listing 2: Implementation of function `foo()` in the source file `a.c`

```
#define BAR false
#include "A.h"

void foo()
//@ requires false;
//@ ensures true;
{
  void **p;
  *p = 0;
}
```

- The source file `b.c`, shown in Listing 3, makes use of the function `foo()`. Before the function `main()` is implemented, the preprocessor symbol `BAR` is defined as `true` and the header file `a.h` is included. By defining `BAR` as `true`, the contract of the forward declaration of function `foo()`, included from `a.h`, indicates that the function `foo()` requires and ensures nothing, but does not contain any errors. So the function `main()` can safely call the function `foo()` in its body.

Listing 3: Implementation of function `main()` in the source file `b.c`

```
#define BAR true
#include "A.h"

int main() // : main
//@ requires true;
//@ ensures true;
{
  foo();
  return 0;
}
```

- It is clear that both `a.c` and `b.c` type check and verify correctly in separation, but the function `foo()` has a different meaning for both files.
- As already mentioned, VeriFast creates a manifest file during the verification of a C source file. The manifest file generated for `a.c` indicates that the latter file implements the function `foo()` that was forward declared in the file `a.h`. The manifest file generated for `b.c`, on the other hand, indicates that the latter file requires the function `foo()` that was forward declared in the file `a.h`. While the symbolic linking process of VeriFast would conclude that the files can be safely linked together (the process only looks at function names and header files containing forward declarations, not at contracts), it is not safe to do so: the resulting application definitely contains a memory access violation.

The problem was shown here for the contract of a function inside a header file. The same problem holds if macro symbols are used for function names or function parameters or other parts of declarations. What they have in common is that to determine the meaning of such a declaration in a header file, the macro context before the include is important. There are different possible solutions to circumvent this problem:

- Do not only include the function signatures in the manifest file, but also the function contracts.
  - The manifest file becomes bloated. To specify the contract of a function, VeriFast allows inductive datatypes, primitive recursive pure functions over these datatypes and abstract separation logic predicates to be defined and used in the contract. These constructs also have to be included in the manifest files, together with all the constructs on which their definition recursively depends.
- Redo the verification of the source files during symbolic linking
  - In many cases this solution is unacceptable (e.g. it deteriorates modularity) or even impossible (e.g. linking with a library when only the header files of that library are available and not the source code).
- Limit the capabilities of the preprocessor so that a header always expands in a safe way.

– The modified (context-free) preprocessor does the trick by processing each included header with an empty set of defined macros. By expanding a header with an empty set of defined macros, a header include always results in the same result. Thus the inclusion is not dependent on the context in which the include occurs (i.e. context-free).

– The context-free preprocessor can then be executed in parallel with the normal preprocessor and if their outputs diverge, the process must be stopped (see Section 7). This ensures that a correct execution (the process is not stopped before completion) of the resulting preprocessor is context-free and compliant with the normal C preprocessor

– This is the solution that is presented in the rest of this text.

## 7 Preprocessing for Sound Symbolic Linking

In this section we will discuss how to limit the capabilities of the C preprocessor to support sound symbolic linking. First the concept of context-free preprocessing is explained. Then, we described how this technique can be used to create a preprocessor that ensures sound symbolic linking.

Before defining the context-free and parallel preprocessing process, we need the auxiliary functions $\mathrm{I}_h$ and $\mathrm{I}_\tau$. The function $\mathrm{I}_\tau$ from Definition 7 expects as input a preprocessor tree $\tau$ and creates a set of all the include nodes in $\tau$. The function $\mathrm{I}_h$ from Definition 8 is very similar. It also expects as input a preprocessor tree $\tau$ and creates a set of all header names in $\tau$. Note that we use a form of overloading and define both functions for a set of include nodes as input as well. Lemma 3 and Lemma 4 state some properties of these definitions which we will need later on.

**Definition 7.**

$$\begin{array}{rcl} I_\tau([]) & = & \varnothing \\ I_\tau(b :: \tau) & = & I_\tau(\tau) \\ I_\tau((h,\tau_h) :: \tau) & = & \{(h,\tau_h)\} \cup I_\tau(\tau_h) \cup I_\tau(\tau) \\ & and & \\ I_\tau(\tilde{\tilde{h}}) & = & \bigcup\limits_{\tilde{h}\in\tilde{\tilde{h}}} I_\tau([\tilde{h}]) \end{array}$$

**Lemma 3.**
$\forall\, \tau.\ I_\tau(\tau) = I_\tau(I_\tau(\tau))$

*Proof.*
We prove this by performing structural induction on $\tau$.

- $\underline{\tau = []}$
  This means $\mathrm{I}_\tau(\mathrm{I}_\tau(\tau)) = \mathrm{I}_\tau(\mathrm{I}_\tau([])) = \mathrm{I}_\tau(\varnothing) = \varnothing = \mathrm{I}_\tau([]) = \mathrm{I}_\tau(\tau)$.

- $\underline{\tau = b :: \tau_r}$
  This means $\mathrm{I}_\tau(\tau) = \mathrm{I}_\tau(b :: \tau_r) = \mathrm{I}_\tau(\tau_r)$. By induction we have:

  - $\mathrm{I}_\tau(\tau_r) = \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_r))$

  And so $\mathrm{I}_\tau(\mathrm{I}_\tau(\tau)) = \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_r)) = \mathrm{I}_\tau(\tau_r) = \mathrm{I}_\tau(\tau)$.

- $\underline{\tau = (h, \tau_h) :: \tau_r}$
  This means $\mathrm{I}_\tau(\tau) = \mathrm{I}_\tau((h, \tau_h) :: \tau_r) = \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$. By induction we have:

  - $\mathrm{I}_\tau(\tau_h) = \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_h))$
  - $\mathrm{I}_\tau(\tau_r) = \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_r))$

  And so:

$$\begin{array}{rcl}
\mathrm{I}_\tau(\mathrm{I}_\tau(\tau)) & = & \mathrm{I}_\tau(\{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)) \\
& = & \mathrm{I}_\tau(\{(h, \tau_h)\}) \cup \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_h)) \cup \mathrm{I}_\tau(\mathrm{I}_\tau(\tau_r)) \\
& = & \mathrm{I}_\tau(\{(h, \tau_h)\}) \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r) \\
& = & \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r) \\
& = & \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r) \\
& = & \mathrm{I}_\tau(\tau)
\end{array}$$

□

**Definition 8.**

$$\begin{array}{rcl}
I_h([]) & = & \varnothing \\
I_h(b :: \tau) & = & I_h(\tau) \\
I_h((h, \tau_h) :: \tau) & = & \{h\} \cup I_h(\tau_h) \cup I_h(\tau) \\
& \textit{and} & \\
I_h(\bar{\tilde{h}}) & = & \bigcup_{\tilde{h} \in \bar{\tilde{h}}} I_h([\tilde{h}])
\end{array}$$

**Lemma 4.**
$\forall\, \tau.\ I_h(\tau) = I_h(I_\tau(\tau))$

*Proof.*
We prove this by performing structural induction on $\tau$.

- $\underline{\tau = []}$
  This means $\mathrm{I}_h([]) = \varnothing = \mathrm{I}_\tau([]) = \mathrm{I}_h(\mathrm{I}_\tau([]))$.

- $\underline{\tau = b :: \tau_r}$
  This means $\mathrm{I}_h(b :: \tau_r) = \mathrm{I}_h(\tau_r)$ and $\mathrm{I}_h(\mathrm{I}_\tau(b :: \tau_r)) = \mathrm{I}_h(\mathrm{I}_\tau(\tau_r))$. By induction we have:

  - $\mathrm{I}_h(\tau_r) = \mathrm{I}_h(\mathrm{I}_\tau(\tau_r))$

  And so $\mathrm{I}_h(b :: \tau_r) = \mathrm{I}_h(\mathrm{I}_\tau(b :: \tau_r))$.

- $\underline{\tau = (h, \tau_h) :: \tau_r}$
  This means $\mathrm{I}_h((h, \tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_r)$ and $\mathrm{I}_h(\mathrm{I}_\tau((h, \tau_h) :: \tau_r)) = \mathrm{I}_h(\{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r))$. By induction we have:

  - $\mathrm{I}_h(\tau_h) = \mathrm{I}_h(\mathrm{I}_\tau(\tau_h))$

– $\mathrm{I}_h(\tau_r) = \mathrm{I}_h(\mathrm{I}_\tau(\tau_r))$

And so:

$$\begin{array}{rcl}
\mathrm{I}_h(\mathrm{I}_\tau((h, \tau_h) :: \tau_r)) & = & \mathrm{I}_h(\{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)) \\
 & = & \mathrm{I}_h(\{(h, \tau_h)\}) \cup \mathrm{I}_h(\mathrm{I}_\tau(\tau_h)) \cup \mathrm{I}_h(\mathrm{I}_\tau(\tau_r)) \\
 & = & \mathrm{I}_h(\{(h, \tau_h)\}) \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_r) \\
 & = & \{h\} \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_r) \\
 & = & \{h\} \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_r) \\
 & = & \mathrm{I}_h((h, \tau_h) :: \tau_r)
\end{array}$$

□

## 7.1 Context-free preprocessing

Here, a modified version of the preprocessor will be defined. It allows sound symbolic linking, but behaves different from the preprocessor from Section 4. The inference rules for the context-free preprocessor are very similar to those from Definition 2. In fact, the following reduction rules are the same and are omitted from Definition 9:

- CFP-WORD-EMPTY = P-WORD-EMPTY
- CFP-WORD-UNDEFINED = P-WORD-UNDEFINED
- CFP-WORD-DEFINED = P-WORD-DEFINED
- CFP-SEQUENCE = P-SEQUENCE
- CFP-DEFINE = P-DEFINE
- CFP-UNDEFINE = P-UNDEFINE
- CFP-BRANCH-IF = P-BRANCH-IF
- CFP-BRANCH-ELSE = P-BRANCH-ELSE

The only other inference rule (i.e CFP-HEADER-CF-EXP) is shown in Definition 9 again using big-step semantics. As for the normal preprocessor, the judgment actually defines a function from a specific $m$, $d$ and $t$ to a resulting $d'$ and $\tau$ as specified in Lemma 5. Given the same header map $m$, defined macros $d$ and token $t$, the output of the context-free preprocessor is very similar to that of the normal preprocessor. They are not the same however, since the rule CFP-HEADER-CF-EXP is different from the rule P-HEADER-EXP. For the normal preprocessor, the rule P-HEADER-EXP performs a recursive call with the same set of defined macros. For the context-free preprocessor on the other hand, a recursive call with an empty set of defined macros is performed in the rule CFP-HEADER-CF-EXP. This rule makes sure that a header is always expanded in the same way as stated in Lemma 7, since earlier defined macros do not influence the expansion of a header. So the semantics of an included header cannot differ among include sites and sound symbolic linking is achieved by using this preprocessor.

**Definition 9.**

$$\frac{m \vdash (\varnothing, m(h)) \Downarrow_{cf} (d', \tau)}{m \vdash (d, h) \Downarrow_{cf} (d \cup d', [(h, \tau)])} \text{ CFP-}\textsc{header-cf-exp}$$

**Lemma 5.**

$$\begin{aligned}\forall\, m, d, d'_1, d'_2, t, \tau_{p_1}, \tau_{p_1}.\ (m \vdash (d, t) \Downarrow_{cf} (d'_1, \tau_{p_1}) \wedge\\ m \vdash (d, t) \Downarrow_{cf} (d'_2, \tau_{p_2})) \Rightarrow d'_1 = d'_2 \wedge \tau_{p_1} = \tau_{p_2}\end{aligned}$$

*Proof.*
Very similar to proof of Lemma 1. □

**Lemma 6.**

$$\begin{aligned}\forall m, d, d', t, \tau_{cfp}.\ m \vdash (d, t) \Downarrow_{cf} (d', \tau_{cfp}) \Rightarrow\\ (\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in I_\tau(\tau_{cfp}) \wedge (h, \tau_2) \in I_\tau(\tau_{cfp})) \Rightarrow \tau_1 = \tau_2)\end{aligned}$$

*Proof.*
It is clear from Definition 7 that the only way to get an include node $(h, \tau_h)$ as an element of $\mathrm{I}_\tau(\tau_{cfp})$, is that it must be a node in $\tau_{cfp}$. But then, the only way to get that include node $(h, \tau_h)$ somewhere in $\tau_{cfp}$, is to use the conclusion of rule CFP-HEADER-CF-EXP from Definition 9. Since for all $h$ (assuming $m$ is fixed), there exists unique $d'$ and $\tau$ such that $m \vdash (\varnothing, m(h)) \Downarrow (d', \tau)$ according to Lemma 5, the premise of rule CFP-HEADER-CF-EXP ensures that forall $\tau_1$ and $\tau_2$ such that $(h, \tau_1) \in \mathrm{I}_\tau(\tau_{cfp}) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_{cfp})$ it must be true that $\tau_1 = \tau_2$. □

**Lemma 7.**

$$\begin{aligned}\forall m, d, d', t, \tau_{cfp}.\ m \vdash (d, t) \Downarrow_{cf} (d', \tau_{cfp}) \Rightarrow\\ (\forall h, \tau_h.\ (h, \tau_h) \in I_\tau(\tau_{cfp}) \Rightarrow (h, \tau_h) \notin I_\tau(\tau_h))\end{aligned}$$

*Proof.*
For the sake of contradiction assume that for a given derivation $m \vdash (d, t) \Downarrow_{cf} (d', \tau_{cfp})$, there exists a header node $(h, \tau_h)$ in the tree $\tau_{cfp}$ (i.e. $(h, \tau_h) \in \mathrm{I}_\tau(\tau_{cfp})$) such that $(h, \tau_h) \in \mathrm{I}_\tau(\tau_h)$. This means that there is a path in the tree $\tau_{cfp}$ from node $(h, \tau_h)$ to another node $(h, \tau_h)$. But since $\tau_h = \tau_h$ there is an infinite path containing infinite many $(h, \tau_h)$ nodes. This contradicts with the finite derivation of $m \vdash (d, t) \Downarrow_{cf} (d', \tau_{cfp})$.

□

## 7.2 Parallel preprocessing

Up until now we have described two different forms of preprocessing: normal preprocessing and context-free preprocessing. Normal preprocessing is what we want because it conforms to the C preprocessor, but we also want context-free preprocessing because it supports sound symbolic linking. The solution to this apparently contradictory problem is to run both preprocessors in parallel and fail if they do not agree on a certain input.

## 7.3 Header guards

Care must be taken with the inclusion of headers files. Since the inference rule CPF-HEADER-EXP-REC, calls the preprocessor recursively with an empty set of defined macros, the macro guarding a header file is never defined at that point during preprocessing. Thus the second time a guarded header is included, it is expanded anyway by the context-free preprocessor. The normal preprocessor will not expand the second include of that guarded header. So, the parallel preprocessing technique described in Subsection 7.2 will fail here.

To make parallel preprocessing succeed for guarded headers, the secondary occurrences of header includes during context-free preprocessing must be ignored when checking if then normal and the context-free preprocessor agree on a certain input. This is safe to do, since the context-free preprocessor always expands a header to the same parse tree (see Lemma 6).

Lets now formalize the parallel preprocessing process that supports guarded headers. For this we need the function RSO which removes the secondary occurrences of header includes from a given preprocessor tree. This function is fairly straightforward specified in Definition 10 where the set $\bar{h}$ is used to keep track of already encountered headers. Lemma 8 states a property of the function RSO that will be necessary later on.

**Definition 10.**

$$\begin{array}{rcll}
RSO([], \bar{h}) & = & [] & \\
RSO(b :: \tau, \bar{h}) & = & b :: RSO(\tau, \bar{h}) & \\
RSO((h, \tau_h) :: \tau, \bar{h}) & = & (h, []) :: RSO(\tau, \bar{h}) & (\textit{if } h \in \bar{h}) \\
RSO((h, \tau_h) :: \tau, \bar{h}) & = & \textbf{let}\, \tau_{h'} = RSO(\tau_h, \bar{h} \cup \{h\})\, \textbf{in} & (\textit{if } h \notin \bar{h}) \\
 & & (h, \tau_{h'}) :: RSO(\tau, \bar{h} \cup I_h([(h, \tau_{h'})])) &
\end{array}$$

**Lemma 8.**

$$\forall \bar{\bar{h}}, \tau. \left\{ \begin{array}{l} \bar{\bar{h}} = I_\tau(\bar{\bar{h}}) \wedge \\ (\forall h, \tau_1, \tau_2.\ \ ((h, \tau_1) \in I_\tau(\tau) \wedge (h, \tau_2) \in I_\tau(\tau) \cup \bar{\bar{h}}) \Rightarrow \tau_1 = \tau_2) \end{array} \right.$$
$$\Rightarrow I_h(\bar{\bar{h}}) \cup I_h(RSO(\tau, I_h(\bar{\bar{h}}))) = I_h(\bar{\bar{h}}) \cup I_h(\tau)$$

*Proof.*
We prove this by performing structural induction on $\tau$.

- $\underline{\tau = []}$
  This means $\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = \mathrm{RSO}([], \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = [] = \tau$ and so definitely:

$$\begin{array}{rcl} \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}([], \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h([]) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\tau) \end{array}$$

- $\underline{\tau = b :: \tau_r}$
  This means $\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = \mathrm{RSO}(b :: \tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = b :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))$. Since we have:

  - $\bar{\bar{\tilde{h}}} = \mathrm{I}_\tau(\bar{\bar{\tilde{h}}})$
    (assumption)
  - $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\bar{\tilde{h}}}) \Rightarrow \tau_1 = \tau_2$
    (strengthening of premise of assumption)

  we know by induction

$$\mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) = \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\tau_r) \tag{1}$$

  and so we have:

$$\begin{array}{rcl} \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(b :: \tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(b :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\tau_r) \\ & & \text{(using Equation 1)} \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(b :: \tau_r) \\ & = & \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\tau) \end{array}$$

- $\underline{\tau = (h, \tau_h) :: \tau_r}$ and $(h, \tau_h) \in \mathrm{I}_\tau(\bar{\bar{\tilde{h}}})$
  This means $h \in \mathrm{I}_h(\bar{\bar{\tilde{h}}})$ and so $\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = \mathrm{RSO}((h, \tau_h) :: \tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}})) = (h, []) :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))$. Since we have:

  - $\bar{\bar{\tilde{h}}} = \mathrm{I}_\tau(\bar{\bar{\tilde{h}}})$
    (assumption)
  - $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\bar{\tilde{h}}}) \Rightarrow \tau_1 = \tau_2$
    (strengthening of premise of assumption)

  we know by induction:

$$\mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}))) = \mathrm{I}_h(\bar{\bar{\tilde{h}}}) \cup \mathrm{I}_h(\tau_r) \tag{2}$$

and so finally:

$$\begin{aligned}
\mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{h}}))) &= \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h(\mathrm{RSO}((h, \tau_h) :: \tau_r, \mathrm{I}_h(\bar{\bar{h}}))) \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h((h, []) :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}))) \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \{h\} \cup \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}))) \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \{h\} \cup \mathrm{I}_h(\tau_r) \\
&\quad \text{(using Equation 2)} \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \{h\} \cup \mathrm{I}_h(\tau_h) \cup \mathrm{I}_h(\tau_r) \\
&\quad \text{(since } (h, \tau_h) \in \bar{\bar{h}} \text{ and } \bar{\bar{h}} = \mathrm{I}_\tau(\bar{\bar{h}}), \\
&\quad \text{we know } \mathrm{I}_\tau(\tau_h) \subseteq \mathrm{I}_\tau(\bar{\bar{h}}) \text{ and so} \\
&\quad \mathrm{I}_h(\tau_h) \subseteq \mathrm{I}_h(\bar{\bar{h}})) \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h((h, \tau_h) :: \tau_r) \\
&= \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h(\tau)
\end{aligned}$$

- $\underline{\tau = (h, \tau_h) :: \tau_r}$ and $(h, \tau_h) \notin \mathrm{I}_\tau(\bar{\bar{h}})$

  For the sake of contradiction, suppose $h \in \mathrm{I}_h(\bar{\bar{h}})$ or equivalently $h \in \bar{\bar{h}}$. This would mean that there exists some $\tau'_h$ such that $(h, \tau'_h) \in \bar{\bar{h}}$. Then, according to

  - $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau) \cup \bar{\bar{h}}) \Rightarrow \tau_1 = \tau_2$

  the equation $\tau_h = \tau'_h$ must hold. But then $(h, \tau_h) \in \bar{\bar{h}}$ which is a contradiction. So we know that $h \notin \mathrm{I}_h(\bar{\bar{h}})$.

  Let $\tau'_h = \mathrm{RSO}(\tau_h, \mathrm{I}_h(\bar{\bar{h}}))$. We now have $\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\bar{h}})) = \mathrm{RSO}((h, \tau_h) :: \tau_r, \mathrm{I}_h(\bar{\bar{h}})) = (h, \tau'_h) :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h([(h, \tau'_h)]))$. Since:

  - $\bar{\bar{h}} = \mathrm{I}_\tau(\bar{\bar{h}})$
    (assumption)
  - $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_h) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}) \Rightarrow \tau_1 = \tau_2$
    (strengthening of premise of assumption)

  we know by induction:

$$\mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h(\tau'_h) = \mathrm{I}_h(\bar{\bar{h}}) \cup \mathrm{I}_h(\tau_h) \tag{3}$$

  We also know:

  - $\mathrm{I}_\tau(\bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})) = \mathrm{I}_\tau(\bar{\bar{h}}) \cup \mathrm{I}_\tau(\mathrm{I}_\tau(\{(h, \tau_h)\})) = \bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})$
  - $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r)\ \wedge$
    $(h, \tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})) \Rightarrow \tau_1 = \tau_2$
    (strengthening of premise of assumption)

  we have again by induction:

$$\begin{aligned}
&\mathrm{I}_h(\bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})) \cup \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})))) = \\
&\qquad\qquad \mathrm{I}_h(\bar{\bar{h}} \cup \mathrm{I}_\tau(\{(h, \tau_h)\})) \cup \mathrm{I}_h(\tau_r)
\end{aligned} \tag{4}$$

Finally we have:

$$\begin{aligned}
\mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h(\mathrm{RSO}(\tau, \mathrm{I}_h(\bar{\tilde{h}}))) &= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h(\mathrm{RSO}((h, \tau_h) :: \tau_r, \mathrm{I}_h(\bar{\tilde{h}}))) \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h((h, \tau'_h) :: \\
&\quad \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau'_h)]))) \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \{h\} \cup \mathrm{I}_h(\tau'_h) \cup \\
&\quad \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau'_h)]))) \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \{h\} \cup \mathrm{I}_h(\tau_h) \cup \\
&\quad \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau_h)]))) \\
&\quad \text{(using Equation 3)} \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau_h)]) \cup \\
&\quad \mathrm{I}_h(\mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau_h)]))) \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h([(h, \tau_h)]) \cup \mathrm{I}_h(\tau_r) \\
&\quad \text{(using Equation 4 and the fact that} \\
&\quad \mathrm{I}_h([(h, \tau_h)]) = \mathrm{I}_h(\mathrm{I}_\tau([(h, \tau_h)]))\ \text{holds)} \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h((h, \tau_h) :: \tau_r) \\
&= \mathrm{I}_h(\bar{\tilde{h}}) \cup \mathrm{I}_h(\tau)
\end{aligned}$$

□

Let the judgment $m, t \blacktriangleright \tau_p, \tau_{cfp}$ as defined in Definition 11 indicate that parallel preprocessing succeeded and produced the normal preprocessor tree $\tau_p$ and the context-free preprocessor tree $\tau_{cfp}$ for a specific token $t$ and header map $m$ (and an empty set of defined macros). So the implementation of the parallel preprocessing technique (see Section 9) must ensure that $m, t \blacktriangleright \tau_p, \tau_{cfp}$ holds.

**Definition 11.**

$$\forall\, m, t, \tau_p, \tau_{cfp}.\;\; m, t \blacktriangleright \tau_p, \tau_{cfp} \Leftrightarrow \exists\, d_p, d_{cfp}.\;\; \begin{cases} m \vdash (\varnothing, t) \Downarrow (d_p, \tau_p) \;\; \wedge \\ m \vdash (\varnothing, t) \Downarrow_{cf} (d_{cfp}, \tau_{cfp}) \;\; \wedge \\ \tau_p = RSO(\tau_{cfp}, \varnothing) \end{cases}$$

### 7.3.1 Updated recursive type checking

As mentioned before, to support symbolic linking the type checking procedure must be recursive and the result of this recursive procedure must be presented to the verification process. But now the type checking process must be modified to account for guarded headers.

The function $\mathrm{VF}_s$ represents the updated recursive type checking process which expects a context-free preprocessor tree as input. The output of $\mathrm{VF}_s$ is, as for VF, again a type checking environment and represents the semantics of the corresponding source file as seen by the verification process. Besides a context-free preprocessor tree $\tau$ and a direct type checking environment $e_d$, the function $\mathrm{VF}_s$ also expects a set of transitive encountered header names $\bar{\tilde{h}}_t$. In this map the first occurrences of included headers and all transitive include nodes in the included tree are remembered so they can be looked up later on. This is clear from the rule for $\mathrm{VF}_s(\tilde{h} :: \tau, \bar{\tilde{h}}_t, e_d)$ in Definition 12.

The rule for VF$_s(b :: \tau, \bar{\tilde{h}}_t, e_d)$ does all the work to get the correct recursive type checking environment. Note the subscript $d$ in $e_d$ in this rule. This means that $e_d$ is the direct type checking environment, i.e. it only contains declarations directly declared in the current expansion (or the initial source file). The let expression in the rule for VF$_s(b :: \tau, \bar{\tilde{h}}_t, e_d)$ then calculates the transitive type checking environment $e$ using the auxiliary function MH from Definition 13. From this definition it follows that $e$ contains, besides the type checked declarations in $e_d$, all the declarations occurring in the header nodes in $\bar{\tilde{h}}_t$, where each header node is type checked with an empty set of included header nodes and an empty type checking environment. This transitive environment $e$ is then used to type check $b$ and the resulting type checked declaration block is added to the current type checking environment before the rest of the tree is processed.

**Definition 12.**

$$\begin{array}{rcl} VF_s([], \bar{\tilde{h}}_t, e_d) & = & e_d \\ VF_s(b :: \tau, \bar{\tilde{h}}_t, e_d) & = & \textbf{let } e := e_d \uplus MH(\bar{\tilde{h}}_t) \textbf{ in} \\ & & VF_s(\tau, \bar{\tilde{h}}_t, e_d \uplus \{|(b, e)|\}) \\ VF_s(\tilde{h} :: \tau, \bar{\tilde{h}}_t, e_d) & = & VF_s(\tau, \bar{\tilde{h}}_t \cup I_\tau([\tilde{h}]), e_d) \end{array}$$

**Definition 13.**

$$MH(\bar{\tilde{h}}) \quad = \quad \biguplus_{(h,\tau) \in \bar{\tilde{h}}} \; VF_s(\tau, \varnothing, \varnothing)$$

Similar as in Subsection 5.4, we can now try to prove that for the two preprocessor trees $\tau_p$ and $\tau_{cfp}$ (if parallel preprocessing succeeded) the type checking environment calculated by CP is equivalent to that calculated by VF$_s$. Since this proof is somewhat more complicated and thus more elaborate, it is covered in the next section.

## 8 Equivalence Proof

We will now prove that for any outputted preprocessor tree $\tau$ the type checking environment calculated by CP is equivalent to that calculated by VF$_s$. This ensures that the semantics of recursive type checking are the same as those of normal type checking if parallel preprocessing is used. Theorem 2 states this more formally.

**Theorem 2.**

$$\forall\, m, t, \tau_p, \tau_{cfp}.\;\; m, t \blacktriangleright \tau_p, \tau_{cfp} \;\; \Rightarrow \;\; CP(\tau_p, \varnothing) \overset{\sim}{\approx} VF_s(\tau_{cfp}, \varnothing, \varnothing) \uplus MH(I_\tau(\tau_{cfp}))$$

As in Subsection 5.5, we need some auxiliary lemmas to prove Theorem 2. The lemmas will now be proven in turn.

## 8.1 Main Lemma

**Lemma 9.**

$$\forall \tau_p, \tau_{cfp}, e_g, e_d, e_o, \bar{\tilde{h}}_t, \bar{\tilde{h}}_o.\ \begin{cases} \bar{\tilde{h}}_t = I_\tau(\bar{\tilde{h}}_t) \wedge \\ \bar{\tilde{h}}_o = I_\tau(\bar{\tilde{h}}_o) \wedge \\ (\forall h, \tau_h.\ \ (h, \tau_h) \in I_\tau(\tau_{cfp}) \ \Rightarrow (h, \tau_h) \notin I_\tau(\tau_h)) \wedge \\ (\forall h, \tau_1, \tau_2.\ \ ((h, \tau_1) \in I_\tau(\tau_{cfp}) \wedge \\ \qquad\qquad (h, \tau_2) \in I_\tau(\tau_{cfp}) \cup \bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \Rightarrow \tau_1 = \tau_2) \wedge \\ \tau_p = \mathit{RSO}(\tau_{cfp}, I_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) \wedge \\ e_g \succapprox e_d \uplus \mathit{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \uplus e_o \end{cases}$$

$$\Rightarrow\ \mathit{CP}(\tau_p, e_g) \succapprox \mathit{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) \uplus \mathit{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup I_\tau(\tau_{cfp})) \uplus e_o$$

*Proof.*
We prove Lemma 9 by performing structural induction on $\tau_{cfp}$.

The only base case for this induction:

- $\underline{\tau_{cfp} = []}$

  From $\tau_p = \mathrm{RSO}([], \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = []$, it follows that:

$$\begin{aligned} \mathrm{CP}(\tau_p, e_g) &= \mathrm{CP}([], e_g) \\ &= e_g \\ \mathrm{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) &= \mathrm{VF}_s([], \bar{\tilde{h}}_t, e_d) \\ &= e_d \end{aligned}$$

  Since $e_g \succapprox e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \uplus e_o$ is given and we know $\mathrm{I}_\tau([]) = \varnothing$ according to Definition 7,

$$\mathrm{CP}(\tau_p, e_g) \succapprox \mathrm{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau(\tau_{cfp})) \uplus e_o$$

  follows immediately.

There are four inductive cases (the case for $\tau_{cfp} = (h, \tau_h) :: \tau_r$ is split into three subcases):

- $\underline{\tau_{cfp} = b :: \tau_r}$

  Let $\tau_r' = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o))$, then $\tau_p = \mathrm{RSO}(b :: \tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = b :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = b :: \tau_r'$, so we have:

$$\begin{aligned} \mathrm{CP}(\tau_p, e_g) &= \mathrm{CP}(b :: \tau_r', e_g) \\ &= \mathrm{CP}(\tau_r', e_g \uplus \{\!|(b, e_g)|\!\}) \\ \mathrm{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) &= \mathrm{VF}_s(b :: \tau_r, \bar{\tilde{h}}_t, e_d) \\ &= \mathrm{VF}_s(\tau_r, \bar{\tilde{h}}_t, e_d \uplus \{\!|(b, e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t))|\!\}) \end{aligned} \tag{5}$$

  and we need to prove:

$$\mathrm{CP}(\tau_p, e_g) \succapprox \mathrm{VF}_s(b :: \tau_r, \bar{\tilde{h}}_t, e_d) \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau(b :: \tau_r)) \uplus e_o \tag{6}$$

Since $e_g \gtrapprox e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \uplus e_o$ is given, and it is clear from Definition 13 that $\mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) = \mathrm{MH}(\bar{\tilde{h}}_t) \uplus \mathrm{MH}(\bar{\tilde{h}}_o \setminus \bar{\tilde{h}}_t)$, we know according to Definition 5 and Definition 6:

$$e_g \uplus \{|(b, e_g)|\} \gtrapprox e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \uplus e_o \uplus \{|(b, e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t))|\}$$

We now have collected the following facts:

- $\bar{\tilde{h}}_t = \mathrm{I}_\tau(\bar{\tilde{h}}_t)$ (assumption)
- $\bar{\tilde{h}}_o = \mathrm{I}_\tau(\bar{\tilde{h}}_o)$ (assumption)
- $\forall h, \tau_h.\ (h, \tau_h) \in \mathrm{I}_\tau(\tau_r) \Rightarrow (h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)$
  (from assumption since $\mathrm{I}_\tau(b :: \tau_r) = \mathrm{I}_\tau(\tau_r)$ according to Definition 8)
- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \Rightarrow \tau_1 = \tau_2$
  (same as previous argument)
- $\tau'_r = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o))$
- $e_g \uplus \{|(b, e_g)|\} \gtrapprox e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o) \uplus e_o \uplus \{|(b, e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t))|\}$

so after applying the induction hypothesis we get:

$$\begin{aligned}&\mathrm{CP}(\tau'_r, e_g \uplus \{|(b, e_g)|\}) \gtrapprox \\ &\qquad \mathrm{VF}_s(\tau_r, \bar{\tilde{h}}_t, e_d \uplus \{|(b, e_d \uplus \mathrm{MH}(\bar{\tilde{h}}_t))|\}) \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau(\tau_r)) \uplus e_o\end{aligned}$$

which we can rewrite to the required environment equivalence relationship (Equation 6) since $\mathrm{I}_\tau(b :: \tau_r) = \mathrm{I}_\tau(\tau_r)$ according to Definition 8 and by using Equation 5:

$$\mathrm{CP}(\tau_p, e_g) \gtrapprox \mathrm{VF}_s(b :: \tau_r, \bar{\tilde{h}}_t, e_d) \uplus \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau(b :: \tau_r)) \uplus e_o$$

- $\underline{\tau_{cfp} = (h, \tau_h) :: \tau_r}$ and $(h, \tau_h) \in \bar{\tilde{h}}_t$

From the assumption $\bar{\tilde{h}}_t = \mathrm{I}_\tau(\bar{\tilde{h}}_t)$ it now follows that $(h, \tau_h) \in \mathrm{I}_\tau(\bar{\tilde{h}}_t)$ and thus $h \in \mathrm{I}_h(\bar{\tilde{h}}_t)$ according to Definition 7 and Definition 8. So definitely $h \in \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)$.

Let $\tau'_r = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o))$, Then $\tau_p = \mathrm{RSO}((h, \tau_h) :: \tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = (h, []) :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = (h, []) :: \tau'_r$ and so we have:

$$\begin{aligned}\mathrm{CP}(\tau_p, e_g) &= \mathrm{CP}((h, []) :: \tau'_r, e_g) \\ &= \mathrm{CP}(\tau'_r, \mathrm{CP}([], e_g)) \\ &= \mathrm{CP}(\tau'_r, e_g) \\ \mathrm{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) &= \mathrm{VF}_s((h, \tau_h) :: \tau_r, \bar{\tilde{h}}_t, e_d) \\ &= \mathrm{VF}_s(\tau_r, \bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]), e_d)\end{aligned} \tag{7}$$

and we need to prove:

$$\begin{aligned}&\mathrm{CP}(\tau_p, e_g) \gtrapprox \mathrm{VF}_s((h, \tau_h) :: \tau_r, \bar{\tilde{h}}_t, e_d) \uplus \\ &\qquad\qquad \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau((h, \tau_h) :: \tau_r)) \uplus e_o\end{aligned} \tag{8}$$

We have collected following facts:

- $\bar{\bar{h}}_t = \mathrm{I}_\tau(\bar{\bar{h}}_t)$ (assumption)
- $\bar{\bar{h}}_o = \mathrm{I}_\tau(\bar{\bar{h}}_o)$ (assumption)
- $\forall h, \tau_h.\ (h, \tau_h) \in \mathrm{I}_\tau(\tau_r) \Rightarrow (h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)$
  (by strengthening of premise of assumption since $\mathrm{I}_\tau((h, \tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$ according to Definition 8)
- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \Rightarrow \tau_1 = \tau_2$
  (same as previous argument)
- $\tau_r' = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o))$
- $e_g \gtrapprox e_d \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \uplus e_o$ (assumption)

so after applying the induction hypothesis we get:

$$\mathrm{CP}(\tau_r', e_g) \gtrapprox \mathrm{VF}_s(\tau_r, \bar{\bar{h}}_t, e_d) \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau(\tau_r)) \uplus e_o \tag{9}$$

Since $(h, \tau_h) \in \bar{\bar{h}}_t$ and $\mathrm{I}_\tau(\bar{\bar{h}}_t) = \bigcup_{\tilde{h} \in \bar{\bar{h}}_t} \mathrm{I}_\tau([\tilde{h}])$, we know

$$\mathrm{I}_\tau([(h, \tau_h)]) \subseteq \mathrm{I}_\tau(\bar{\bar{h}}_t)$$

but then since $\bar{\bar{h}}_t = \mathrm{I}_\tau(\bar{\bar{h}}_t)$, we have $\mathrm{I}_\tau([(h, \tau_h)]) \subseteq \bar{\bar{h}}_t$ and thus:

$$\bar{\bar{h}}_t = \bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \tag{10}$$

Since $\mathrm{I}_\tau([(h, \tau_h)]) = \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h)$, we now have:

$$\begin{aligned}
\bar{\bar{h}}_o \cup \bar{\bar{h}}_t \cup \mathrm{I}_\tau(\tau_r) &= \bar{\bar{h}}_o \cup \bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \mathrm{I}_\tau(\tau_r) \\
&\qquad \text{(using Equation 10)} \\
&= \bar{\bar{h}}_o \cup \bar{\bar{h}}_t \cup \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r) \\
&= \bar{\bar{h}}_o \cup \bar{\bar{h}}_t \cup \mathrm{I}_\tau((h, \tau_h) :: \tau_r) \\
&\qquad \text{(using Definition 7)}
\end{aligned} \tag{11}$$

Using these two equalities (Equation 10 and Equation 11) we can rewrite the result of applying the induction hypothesis (Equation 9):

$$\begin{aligned}
\mathrm{CP}(\tau_r', e_g) \gtrapprox\ & \mathrm{VF}_s(\tau_r, \bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]), e_d) \uplus \\
& \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_0 \cup \mathrm{I}_\tau((h, \tau_h) :: \tau_r)) \uplus e_o
\end{aligned}$$

which we can rewrite (using Equation 7) to the required environment equivalence relationship (Equation 8):

$$\begin{aligned}
\mathrm{CP}(\tau_p, e_g) \gtrapprox\ & \mathrm{VF}_s((h, \tau_h) :: \tau_r, \bar{\bar{h}}_t, e_d) \uplus \\
& \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau((h, \tau_h) :: \tau_r)) \uplus e_o
\end{aligned}$$

- $\underline{\tau_{cfp} = (h, \tau_h) :: \tau_r}$ and $(h, \tau_h) \notin \bar{\bar{h}}_t \wedge (h, \tau_h) \in \bar{\bar{h}}_o$

From the assumption $\bar{\bar{h}}_o = \mathrm{I}_\tau(\bar{\bar{h}}_o)$ it now follows that $(h, \tau_h) \in \mathrm{I}_\tau(\bar{\bar{h}}_o)$ and thus $h \in \mathrm{I}_h(\bar{\bar{h}}_o)$ according to Definition 7 and Definition 8. So definitely $h \in \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)$.

Let $\tau_r' = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o))$, Then $\tau_p = \mathrm{RSO}((h,\tau_h) :: \tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = (h, []) :: \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o)) = (h, []) :: \tau_r'$ and so we have:

$$\begin{array}{rcl} \mathrm{CP}(\tau_p, e_g) & = & \mathrm{CP}((h,[]) :: \tau_r', e_g) \\ & = & \mathrm{CP}(\tau_r', \mathrm{CP}([], e_g)) \\ & = & \mathrm{CP}(\tau_r', e_g) \\ \mathrm{VF}_s(\tau_{cfp}, \bar{\tilde{h}}_t, e_d) & = & \mathrm{VF}_s((h,\tau_h) :: \tau_r, \bar{\tilde{h}}_t, e_d) \\ & = & \mathrm{VF}_s(\tau_r, \bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]), e_d)) \end{array} \tag{12}$$

and we need to prove:

$$\mathrm{CP}(\tau_p, e_g) \succapprox \mathrm{VF}_s((h,\tau_h) :: \tau_r, \bar{\tilde{h}}_t, e_d) \uplus \\ \mathrm{MH}(\bar{\tilde{h}}_t \cup \bar{\tilde{h}}_o \cup \mathrm{I}_\tau((h,\tau_h) :: \tau_r)) \uplus e_o \tag{13}$$

Since $(h,\tau_h) \in \bar{\tilde{h}}_o$ and $\mathrm{I}_\tau(\bar{\tilde{h}}_o) = \bigcup_{\tilde{h} \in \bar{\tilde{h}}_o} \mathrm{I}_\tau([\tilde{h}])$, we know

$$\mathrm{I}_\tau([(h,\tau_h)]) \subseteq \mathrm{I}_\tau(\bar{\tilde{h}}_o)$$

but then since $\bar{\tilde{h}}_o = \mathrm{I}_\tau(\bar{\tilde{h}}_o)$, we have $\mathrm{I}_\tau([(h,\tau_h)]) \subseteq \bar{\tilde{h}}_o$ and thus:

$$\bar{\tilde{h}}_o = \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\tilde{h}}_o$$

From this and the fact that $\mathrm{I}_\tau((h,\tau_h) :: \tau_r) = \mathrm{I}_\tau([(h,\tau_h)]) \cup \mathrm{I}_\tau(\tau_r)$, we can deduce the following by strengthening the premise of an assumption:

$$\forall h, \tau_1, \tau_2.\ ((h,\tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge \\ (h,\tau_2) \in \mathrm{I}_\tau(\tau_r) \cup \bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\tilde{h}}_o) \Rightarrow \tau_1 = \tau_2 \tag{14}$$

Also note that:

$$\begin{array}{rcl} \mathrm{I}_\tau(\bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)])) & = & \mathrm{I}_\tau(\bar{\tilde{h}}_t) \cup \mathrm{I}_\tau(\mathrm{I}_\tau([(h,\tau_h)])) \\ & = & \mathrm{I}_\tau(\bar{\tilde{h}}_t) \cup \mathrm{I}_\tau([(h,\tau_h)]) \\ & & \quad \text{(using Lemma 3)} \\ & = & \bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \\ & & \quad \text{(using that } \bar{\tilde{h}}_t = \mathrm{I}_\tau(\bar{\tilde{h}}_t)) \end{array} \tag{15}$$

We now have collected the following facts:

- $\bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) = \mathrm{I}_\tau(\bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]))$
  (see Equation 15)
- $\bar{\tilde{h}}_o = \mathrm{I}_\tau(\bar{\tilde{h}}_o)$
  (assumption)
- $\forall h, \tau_h.\ (h,\tau_h) \in \mathrm{I}_\tau(\tau_r) \Rightarrow (h,\tau_h) \notin \mathrm{I}_\tau(\tau_h)$
  (by strengthening of premise of assumption since $\mathrm{I}_\tau((h,\tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$ according to Definition 8)
- $\forall h, \tau_1, \tau_2.\ ((h,\tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h,\tau_2) \in$
  $\mathrm{I}_\tau(\tau_r) \cup \bar{\tilde{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\tilde{h}}_o) \Rightarrow \tau_1 = \tau_2$
  (see Equation 14)

– $\tau'_r = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\bar{h}}_o))$
(from assumption since $\bar{\bar{h}}_o = \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\bar{h}}_o$)

– $e_g \gtrapprox e_d \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\bar{h}}_o) \uplus e_o$
(from assumption since $\bar{\bar{h}}_o = \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\bar{h}}_o$)

so after applying the induction hypothesis we get:

$$\mathrm{CP}(\tau'_r, e_g) \gtrapprox \mathrm{VF}_s(\tau_r, \bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]), e_d) \uplus \\ \mathrm{MH}(\bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]) \cup \bar{\bar{h}}_0 \cup \mathrm{I}_\tau(\tau_r)) \uplus e_o$$

which we can rewrite (using Equation 12 and Definition 7) to the required environment equivalence relationship (Equation 13):

$$\mathrm{CP}(\tau_p, e_g) \gtrapprox \mathrm{VF}_s((h,\tau_h) :: \tau_r, \bar{\bar{h}}_t, e_d) \uplus \\ \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau((h,\tau_h) :: \tau_r)) \uplus e_o$$

- $\underline{\tau_{cfp} = (h,\tau_h) :: \tau_r}$ and $(h,\tau_h) \notin \bar{\bar{h}}_t \wedge (h,\tau_h) \notin \bar{\bar{h}}_o$

From the assumptions $\bar{\bar{h}}_t = \mathrm{I}_\tau(\bar{\bar{h}}_t)$ and $\bar{\bar{h}}_o = \mathrm{I}_\tau(\bar{\bar{h}}_o)$ it now follows that $(h,\tau_h) \notin \mathrm{I}_\tau(\bar{\bar{h}}_t) \cup \mathrm{I}_\tau(\bar{\bar{h}}_o)$ and thus $(h,\tau_h) \notin \mathrm{I}_\tau(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)$ or $(h,\tau_h) \notin \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)$.

Let $\tau'_h = \mathrm{RSO}(\tau_h, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \{h\})$ and let $\tau'_r = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h([(h,\tau'_h)]))$. Then $\tau_p = \mathrm{RSO}((h,\tau_h) :: \tau_r, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)) = (h,\tau'_h) :: \tau'_r$ and so we have:

$$\begin{array}{rcl} \mathrm{CP}(\tau_p, e_g) & = & \mathrm{CP}((h,\tau'_h) :: \tau'_r, e_g) \\ & = & \mathrm{CP}(\tau'_r, \mathrm{CP}(\tau'_h, e_g)) \\ \mathrm{VF}_s(\tau_{cfp}, \bar{\bar{h}}_t, e_d) & = & \mathrm{VF}_s((h,\tau_h) :: \tau_r, \bar{\bar{h}}_t, e_d) \\ & = & \mathrm{VF}_s(\tau_r, \bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h,\tau_h)]), e_d)) \end{array} \tag{16}$$

and we need to prove:

$$\mathrm{CP}(\tau_p, e_g) \gtrapprox \mathrm{VF}_s((h,\tau_h) :: \tau_r, \bar{\bar{h}}_t, e_d) \uplus \\ \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau((h,\tau_h) :: \tau_r)) \uplus e_o \tag{17}$$

Note that from the assumption:

$$\forall h', \tau'_h.\ (h', \tau'_h) \in \mathrm{I}_\tau(\tau_{cfp}) \Rightarrow (h', \tau'_h) \notin \mathrm{I}_\tau(\tau'_h)$$

we know $(h,\tau_h) \notin \mathrm{I}_\tau(\tau_h)$, since clearly $(h,\tau_h) \in \mathrm{I}_\tau(\tau_{cfp})$. But then, from the assumption:

$$\forall h, \tau_1, \tau_2.\ ((h,\tau_1) \in \mathrm{I}_\tau(\tau_{cfp}) \wedge (h,\tau_2) \in \mathrm{I}_\tau(\tau_{cfp}) \cup \bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \Rightarrow \tau_1 = \tau_2$$

we can conclude that there exists no $\tau'_h$ so that $(h,\tau'_h) \in \mathrm{I}_\tau(\tau_h)$ and $\tau'_h \neq \tau_h$. Thus we know that $h \notin \mathrm{I}_h(\tau_h)$ and so:

$$\begin{array}{rcl} \tau'_h & = & \mathrm{RSO}(\tau_h, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \{h\}) \\ & = & \mathrm{RSO}(\tau_h, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)) \end{array} \tag{18}$$

using this last equality and the fact that $\bar{\bar{h}}_t \cup \bar{\bar{h}}_o = \mathrm{I}_\tau(\bar{\bar{h}}_t) \cup \mathrm{I}_\tau(\bar{\bar{h}}_o) = \mathrm{I}_\tau(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)$ and using the following fact:

- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_h) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \Rightarrow \tau_1 = \tau_2$
  (by strengthening of premise of assumption since $\mathrm{I}_\tau((h, \tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$ according to Definition 8)

we can deduce the following using Lemma 8:

$$\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\tau'_h) \quad = \quad \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\tau_h)$$

and so:

$$\begin{array}{rcl}
\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\tau'_h) & = & \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\tau_h) \\
\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \{h\} \cup \mathrm{I}_h(\tau'_h) & = & \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \{h\} \cup \mathrm{I}_h(\tau_h) \\
\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h([(h, \tau'_h)]) & = & \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h([(h, \tau_h)]) \\
\text{(using Definition 8)} & & \text{(using Definition 8)} \\
\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\mathrm{I}_\tau([(h, \tau'_h)])) & = & \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \cup \mathrm{I}_h(\mathrm{I}_\tau([(h, \tau_h)])) \\
\text{(using Lemma 4)} & & \text{(using Lemma 4)} \\
\mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau([(h, \tau'_h)])) & = & \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau([(h, \tau_h)]))
\end{array} \qquad (19)$$

We now have collected the following facts:

- $\varnothing = \mathrm{I}_\tau(\varnothing)$
  (trivial)
- $\bar{\bar{h}}_t \cup \bar{\bar{h}}_o = \mathrm{I}_\tau(\bar{\bar{h}}_t) \cup \mathrm{I}_\tau(\bar{\bar{h}}_o) = \mathrm{I}_\tau(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)$
  (from assumptions)
- $\forall h', \tau'_h.\ (h', \tau'_h) \in \mathrm{I}_\tau(\tau_h) \Rightarrow (h', \tau'_h) \notin \mathrm{I}_\tau(\tau'_h)$
  (by strengthening of premise of assumption since $\mathrm{I}_\tau((h, \tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$ according to Definition 8)
- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_h) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \Rightarrow \tau_1 = \tau_2$
  (same as previous argument)
- $\tau'_h = \mathrm{RSO}(\tau_h, \mathrm{I}_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o))$
  (see Equation 18)
- $e_g \gtrapprox \varnothing \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \uplus e_d \uplus e_o$
  (assumption)

so after applying the induction hypothesis we get:

- $\mathrm{CP}(\tau'_h, e_g) \gtrapprox \mathrm{VF}(\tau_h, \varnothing, \varnothing) \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup \mathrm{I}_\tau(\tau_h)) \uplus e_d \uplus e_o$
- $\mathrm{CP}(\tau'_h, e_g) \gtrapprox e_d \uplus \mathrm{VF}(\tau_h, \varnothing, \varnothing) \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}_o) \uplus e_o$
- $\mathrm{CP}(\tau'_h, e_g) \gtrapprox e_d \uplus \mathrm{MH}(\{(h, \tau_h)\}) \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}_o) \uplus e_o$
  (using Definition 13)
- $\mathrm{CP}(\tau'_h, e_g) \gtrapprox e_d \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \{(h, \tau_h)\} \cup \mathrm{I}_\tau(\tau_h) \cup \bar{\bar{h}}_o) \uplus e_o$
  (using Definition 13 again and the previously collected facts: $(h, \tau_h) \notin \bar{\bar{h}}_t$, $(h, \tau_h) \notin \bar{\bar{h}}_o$ and $(h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)$)
- $\mathrm{CP}(\tau'_h, e_g) \gtrapprox e_d \uplus \mathrm{MH}(\bar{\bar{h}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \bar{\bar{h}}_o) \uplus e_o$

Before applying the induction hypothesis again, we need among others the following equation:

$$\begin{array}{rcl} \tau'_r & = & \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}_t \cup \bar{\bar{\tilde{h}}}_o) \cup \mathrm{I}_h([(h, \tau'_h)])) \\ & = & \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}_t \cup \bar{\bar{\tilde{h}}}_o) \cup \mathrm{I}_h(\mathrm{I}_\tau([(h, \tau'_h)]))) \\ & & \quad \text{(using Lemma 4)} \\ & = & \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}_t \cup \bar{\bar{\tilde{h}}}_o \cup \mathrm{I}_\tau([(h, \tau'_h)]))) \\ & & \quad \text{(using Equation 19)} \\ & = & \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}_t \cup \bar{\bar{\tilde{h}}}_o \cup \mathrm{I}_\tau([(h, \tau_h)]))) \end{array} \tag{20}$$

Also note that:

$$\begin{array}{rcl} \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)])) & = & \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_t) \cup \mathrm{I}_\tau(\mathrm{I}_\tau([(h, \tau_h)])) \\ & = & \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_t) \cup \mathrm{I}_\tau([(h, \tau_h)]) \\ & & \quad \text{(using Lemma 3)} \\ & = & \bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \\ & & \quad \text{(using that } \bar{\bar{\tilde{h}}}_t = \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_t)\text{)} \end{array} \tag{21}$$

We now have collected the following facts:

- $\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) = \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]))$
  (see Equation 21)
- $\bar{\bar{\tilde{h}}}_o = \mathrm{I}_\tau(\bar{\bar{\tilde{h}}}_o)$
  (assumption)
- $\forall h, \tau_h.\ (h, \tau_h) \in \mathrm{I}_\tau(\tau_r) \Rightarrow (h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)$
  (by strengthening of premise of assumption since $\mathrm{I}_\tau((h, \tau_h) :: \tau_r) = \{h\} \cup \mathrm{I}_\tau(\tau_h) \cup \mathrm{I}_\tau(\tau_r)$ according to Definition 8)
- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_r) \wedge (h, \tau_2) \in$
  $\mathrm{I}_\tau(\tau_r) \cup \bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \bar{\bar{\tilde{h}}}_o) \Rightarrow \tau_1 = \tau_2$
  (same as previous argument and some rewriting)
- $\tau'_r = \mathrm{RSO}(\tau_r, \mathrm{I}_h(\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \bar{\bar{\tilde{h}}}_o))$
  (see Equation 20)
- $\mathrm{CP}(\tau'_h, e_g) \succapprox e_d \uplus \mathrm{MH}(\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \bar{\bar{\tilde{h}}}_o) \uplus e_o$

so after applying the induction hypothesis again we get:

$$\begin{array}{l} \mathrm{CP}(\tau'_r, \mathrm{CP}(\tau'_h, e_g)) \succapprox \mathrm{VF}_s(\tau_r, \bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]), e_d) \uplus \\ \qquad\qquad \mathrm{MH}(\bar{\bar{\tilde{h}}}_t \cup \mathrm{I}_\tau([(h, \tau_h)]) \cup \bar{\bar{\tilde{h}}}_o \cup \mathrm{I}_\tau(\tau_r)) \uplus e_o \end{array}$$

which we can rewrite (using Equation 16 and Definition 7) to the required environment equivalence relationship (Equation **??**):

$$\mathrm{CP}(\tau_p, e_g) \succapprox \mathrm{VF}_s((h, \tau_h) :: \tau_r, \bar{\bar{\tilde{h}}}_t, e_d) \uplus \mathrm{MH}(\bar{\bar{\tilde{h}}}_t \cup \bar{\bar{\tilde{h}}}_o \cup \mathrm{I}_\tau((h, \tau_h) :: \tau_r)) \uplus e_o$$

□

## 8.2 Proof of Theorem 2

Armed with Lemma 9 it is fairly easy to prove Theorem 2. First, using Axiom 11 we immediately get:

- $\tau_p = \mathrm{RSO}(\tau_{cfp}, \varnothing)$

Then using Lemma 6 we get:

- $\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_{cfp}) \wedge (h, \tau_2) \in \mathrm{I}_\tau(\tau_{cfp})) \Rightarrow \tau_1 = \tau_2$

and using Lemma 7 we get:

- $\forall h, \tau_h.\ (h, \tau_h) \in \mathrm{I}_\tau(\tau_{cfp}) \Rightarrow (h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)$

Lets restate Lemma 9 for our current $\tau_p$ and $\tau_{cfp}$, choosing $\varnothing$ for $e_g$, $e_d$ and $e_o$, and also choosing $\bar{\bar{h}}_t$ and $\bar{\bar{h}}_o$ to be $\varnothing$:

$$\left\{\begin{array}{l}
\varnothing = \mathrm{I}_\tau(\varnothing)\ \wedge \\
\varnothing = \mathrm{I}_\tau(\varnothing)\ \wedge \\
(\forall h, \tau_h.\ (h, \tau_h) \in \mathrm{I}_\tau(\tau_{cfp})\ \Rightarrow (h, \tau_h) \notin \mathrm{I}_\tau(\tau_h)) \wedge \\
(\forall h, \tau_1, \tau_2.\ ((h, \tau_1) \in \mathrm{I}_\tau(\tau_{cfp}) \wedge \\
\qquad\qquad (h, \tau_2) \in \mathrm{I}_\tau(\tau_{cfp}) \cup \varnothing \cup \varnothing) \Rightarrow \tau_1 = \tau_2) \wedge \\
\tau_p = \mathrm{RSO}(\tau_{cfp}, \mathrm{I}_h(\varnothing \cup \varnothing))\ \wedge \\
\varnothing \gtrapprox \varnothing \uplus \mathrm{MH}(\varnothing \cup \varnothing) \cup \varnothing
\end{array}\right.$$
$$\Rightarrow\ \mathrm{CP}(\tau_p, \varnothing) \gtrapprox \mathrm{VF}_s(\tau_{cfp}, \varnothing, \varnothing) \uplus \mathrm{MH}(\varnothing \cup \varnothing \cup \mathrm{I}_\tau(\tau_{cfp})) \uplus \varnothing$$

Since all the premises are clearly true, we finally get:

- $\mathrm{CP}(\tau, \varnothing) \gtrapprox \mathrm{VF}_s(\tau, \varnothing, \varnothing) \uplus \mathrm{MH}(\mathrm{I}_\tau(\tau_{cfp}))$

# 9 Description of Implementation

The recursive type checking described in Subsection 5.4 was already present in VeriFast to support symbolic linking. Actually, that implementation already did something very similar to the function $\mathrm{VF}_s$ defined in Subsection 7. To overcome the problem that occurs when headers are type checked in isolation and secondary occurrences of guarded headers are removed by the C preprocessor, all headers are lexically analyzed, preprocessed, parsed and type checked in isolation. Only then are the declarations it contains added to the type checking environment of the file that included the header. The unsoundness introduced by preprocessing was solved by only allowing header guards and nothing else of the capabilities of the C preprocessor.

The parallel preprocessing technique from Subsection 7.2 was straightforward to implement in VeriFast. An implementation of the C preprocessor and the context-free preprocessor are run in parallel and an error is reported if their output diverge. If the parallel preprocessor encounters a secondary include, it ignores this include to make sure that $m, t \blacktriangleright \tau_p, \tau_{cfp}$ (according to Definition 11) holds.

If a single header is included many times, the function $\mathrm{VF}_s$ is not very efficient. For every declaration block that needs the header for type checking its declarations, the function $\mathrm{VF}_s$ is recursively called for that header through the function MH. In the actual implementation the result for each header is remembered, so the next time it is needed, it does not have to be computed again.

Since the verification process itself did not had to be updated, the necessary modifications were nicely isolated. Only the preprocessing stage and the type checking stage of the verifier had to be updated. This advantage makes the technique presented in this text suitable for implementation in other verifiers.