# A Type System Equivalent to a Model Checker

Mayur Naik[1] and Jens Palsberg[2]

[1] Stanford University
mhn@cs.stanford.edu
[2] UCLA
palsberg@ucla.edu

**Abstract.** Type systems and model checking are two prevalent approaches to program verification. A prominent difference between them is that type systems are typically defined in a syntactic and modular style whereas model checking is usually performed in a semantic and whole-program style. This difference between the two approaches lends them complementary to each other: type systems are good at explaining why a program was accepted while model checkers are good at explaining why a program was rejected.

We present a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. The model checker is natural and may be instantiated with any finite-state abstraction scheme such as predicate abstraction. The type system which is also parametric type checks exactly those programs that are accepted by the model checker. It uses function types to capture flow sensitivity and intersection and union types to capture context sensitivity. Our result sheds light on the relationship between the two approaches, provides a methodology for studying their relative expressiveness, is a step towards sharing results between them, and motivates synergistic program analyses involving interplay between them.

## 1 Introduction

### 1.1 Background

Type systems and model checking are two prevalent approaches to program verification. It is well known that both approaches are essentially abstract interpretations and are therefore closely related [10, 11]. Despite deep connections, however, a prominent difference between them is that type systems are typically defined in a syntactic and modular style, using one type rule per syntactic construct, whereas model checking is usually performed in a semantic and whole-program style, by exploring the reachable state-space of a model of the program. This difference between type systems and model checking has a significant consequence: it lends the approaches complementary to each other, namely, type systems are better at explaining why a program was accepted whereas model checkers are better at explaining why a program was rejected.

© Springer-Verlag Berlin Heidelberg 2005

A type inference algorithm that accepts a program annotates it with *types* (keywords: syntactic, modular) explaining why it was accepted. The benefits of type annotations are well known: they aid in understanding, modifying, reusing and certifying the program. However, it is often unnatural to explain why a program was rejected by a type inference algorithm, and there is a large body of work on explaining the source of type errors especially in the context of type inference algorithms for languages with higher-order functions like Haskell, Miranda, and ML [46, 24, 6, 14, 44, 9, 19] and, more recently, for languages with concurrency like Java [15, 16].

On the other hand, a model checker that rejects a program provides a *counterexample* which is a program trace (keywords: semantic, whole-program) that explains why the program was rejected. The benefits of counterexamples are well known: they aid in debugging the program. However, it is often unnatural to explain why a program was accepted by a model checker, and several proof systems for model checkers have been devised [39, 31, 38, 21, 43, 32].

This complementary nature of type systems and model checking motivates investigating the relationship between the two approaches, devising a methodology for studying their relative expressiveness, sharing results between them, and designing synergistic program analyses involving an interplay between a type system and a model checker.

## 1.2 Our Result

In this paper, we present a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. In model checking terminology, a safety property is a temporal property whose violation can be witnessed by a finite program trace or, equivalently, by the failure of an assertion at a program point. Our model checker is conventional and may be instantiated with any finite-state abstraction scheme such as predicate abstraction [18]. The type system which is also parametric type checks exactly those programs that are accepted by the model checker. It uses function types to capture flow sensitivity and intersection and union types to capture context sensitivity.

The implications of our result may be summarized as follows:

1. Our work sheds light on the relationship between type systems and model checking. In particular, it shows that the most straightforward form of model checking corresponds to the most complex form of typing.
   Finite-state model checkers routinely associate with each statement $s$ of the program a set of the form:
   $$\{\ \langle \omega_i, \omega_j \rangle \mid \omega_j \in \delta_s(\omega_i)\ \}$$
   where $\omega$ ranges over a finite set of abstract contexts $\Omega$ and $\delta_s : \Omega \rightharpoonup 2^{\Omega}$ is a partial function called the abstract transfer function associated with $s$. Intuitively, the above set says that if $s$ begins executing in abstract context $\omega_i$ then it will finish executing in an abstract context $\omega_j \in \delta_s(\omega_i)$. For example, in model checkers such as SLAM [4], BLAST [22], and MAGIC [7], $\Omega$

represents the set of all valuations to the finite set of predicates with respect to which the predicate abstraction (model) of the program is constructed.

Likewise, our type system assigns to each statement in the program, a finitary polymorphic type of the form:

$$\bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$$

where $A$ and $\forall i \in A : B_i$ are finite. This is the most complex form of typing. Conventional type systems employ restricted cases of this form of typing such as ones requiring $|A| = 1$ (no intersection types) or $\forall i \in A : |B_i| = 1$ (no union types).

2. Our work provides a methodology for studying the relative expressiveness of a type system and a model checker. Our technique for proving the equivalence is novel and general: we have successfully applied it in two additional settings, namely, stack-size analysis [25] and deadline analysis [29] for a class of real-time programs called interrupt-driven programs [35].
3. Our work is a step towards sharing of results between the type systems and model checking communities. The backward direction of our equivalence theorem states that if the model checker accepts a program, then the program is well-typed. We prove this by building a type derivation from the model constructed by the model checker. We thereby obtain a model-checking-based type inference algorithm for our type system.
4. Our work motivates synergistic program analyses involving interplay between a type system and a model checker. The analyses can use types to document correct programs and counterexamples to explain erroneous programs. Moreover, they can be implemented efficiently due to the correspondence between types and models: types already existing in the program or inferred by a type inference algorithm can be used to construct a model for performing model checking, as illustrated in [12, 8], and conversely, a model constructed by a model checker can be used to infer types, as shown in this paper.

## 1.3 Proof Architecture

We present an overview of our technique for proving the equivalence. A typical type soundness theorem states that *well-typed programs do not go wrong* [27]. Usually, *going wrong* is formalized as *getting stuck* in the operational semantics. More formally, for a program $s$, an initial concrete environment $\sigma$, and an initial abstract environment $\omega$, type soundness states that:

If $\langle s, \omega \rangle$ is well-typed then $\langle s, \sigma \rangle$ does not go wrong (in the concrete semantics).

Type checking requires a predefined set of abstractions, namely, the types. Then, the existence of a derivable type judgment implies that the program has the desired property. Model checking, on the other hand, is not concerned with types. It works with a model, that is, an abstract semantics, and can answer questions such as:

$\langle s, \omega \rangle$ does not go wrong (in the abstract semantics).

Model-checking soundness then states that:

If $\langle s, \omega \rangle$ does not go wrong (in the abstract semantics) then
$\langle s, \sigma \rangle$ does not go wrong (in the concrete semantics).

Our equivalence result states that:

$\langle s, \omega \rangle$ is well-typed iff $\langle s, \omega \rangle$ does not go wrong (in the abstract semantics).

We prove the forward direction using a variant of type soundness in which the step relation is the abstract semantics instead of the concrete semantics and we prove the backward direction constructively by building a type derivation from the model constructed by the model checker.

It is important to note that we do not prove the soundness of either the type system or the model checker. Our equivalence result guarantees that the type system is sound iff the model checker is sound but it does not prevent both from being unsound. Proving soundness would require us to define a concrete semantics and to instantiate the type system and the model checker (recall that both are parametric). This in turn would detract from the generality of our equivalence result.

### 1.4 Rest of the Paper

In Section 2, we present an imperative WHILE language and a model checker for verifying temporal safety properties expressed as assertions in that language. In Section 3, we present a type system that is equivalent to the model checker. In Section 4, we prove the equivalence result. In Section 5, we illustrate the equivalence by means of examples. In Section 6, we discuss related work. Finally, in Section 7, we conclude with a note on future work.

## 2 Model Checker

The abstract syntax of our imperative WHILE language is as follows:

$$\text{(stmt)}\ s ::= p \mid \texttt{assume}(e) \mid \texttt{assert}(e) \mid s_1;\ s_2 \mid \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (*)\ \texttt{do}\ s'$$

A statement $s$ is either a primitive statement $p$ (for instance, an assignment statement or a skip statement), an assume statement, an assert statement, a sequential composition of statements, a branching statement, or a looping statement. For the sake of generality, we leave primitive statements $p$ and boolean expressions $e$ uninterpreted. Our abstract syntax for branching and looping statements is standard in the literature on model checking. It is related to the more familiar syntax for these statements as follows:

$$\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \equiv \texttt{if}\ (*)\ \texttt{then}\ \{\, \texttt{assume}(e);\ s_1 \,\}\ \texttt{else}\ \{\, \texttt{assume}(\bar{e});\ s_2 \,\}$$
$$\texttt{while}\ (e)\ \texttt{do}\ s' \equiv \{\, \texttt{while}\ (*)\ \texttt{do}\ \{\, \texttt{assume}(e);\ s' \,\}\,\};\ \texttt{assume}(\bar{e})$$

$$\text{(state)} \quad a \ ::= \ \omega \ \mid \ \mathsf{error} \ \mid \ \langle s, \omega \rangle$$

$$\langle p, \omega_k \rangle \hookrightarrow \omega_l \qquad \text{if } l \in \delta_p(k) \tag{1}$$

$$\langle \texttt{assume}(e), \omega_k \rangle \hookrightarrow \omega_k \qquad \text{if } k \in \delta_e \tag{2}$$

$$\langle \texttt{assume}(e), \omega_k \rangle \hookrightarrow \mathsf{error} \qquad \text{if } k \notin \delta_e \tag{3}$$

$$\langle \texttt{assert}(e), \omega_k \rangle \hookrightarrow \omega_k \qquad \text{if } k \in \delta_e \tag{4}$$

$$\frac{\langle s_1, \omega \rangle \hookrightarrow \omega'}{\langle s_1; s_2, \omega \rangle \hookrightarrow \langle s_2, \omega' \rangle} \ (5) \qquad \frac{\langle s_1, \omega \rangle \hookrightarrow \mathsf{error}}{\langle s_1; s_2, \omega \rangle \hookrightarrow \mathsf{error}} \ (6) \qquad \frac{\langle s_1, \omega \rangle \hookrightarrow \langle s_1', \omega' \rangle}{\langle s_1; s_2, \omega \rangle \hookrightarrow \langle s_1'; s_2, \omega' \rangle} \ (7)$$

$$\langle \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2, \omega \rangle \hookrightarrow \langle s_1, \omega \rangle \tag{8}$$

$$\langle \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2, \omega \rangle \hookrightarrow \langle s_2, \omega \rangle \tag{9}$$

$$\langle \texttt{while } (*) \texttt{ do } s', \omega \rangle \hookrightarrow \langle s'; \texttt{ while } (*) \texttt{ do } s', \omega \rangle \tag{10}$$

$$\langle \texttt{while } (*) \texttt{ do } s', \omega \rangle \hookrightarrow \omega \tag{11}$$

**Fig. 1.** Abstract Semantics

where $(*)$ denotes non-deterministic choice and $\bar{e}$ denotes the negation of $e$.

We next present a model checker for verifying temporal safety properties of programs expressed in our language. The class of temporal safety properties is precisely the class of properties whose violation can be witnessed by a finite program trace or, equivalently, by the failure of an assertion at a program point. Our model checker is conventional and is parameterized by the following components:

– A finite set of abstract contexts $\Omega$.
– An abstract transfer function $\delta_p \in \Omega \to 2^\Omega$ per primitive statement $p$ describing the effect of $p$ on abstract contexts. We assume that $\delta_p$ is total and $\forall i \in \Omega : \delta_p(i) \neq \emptyset$.
– A predicate $\delta_e \subseteq \Omega$ per boolean expression $e$ denoting the set of abstract contexts in which $e$ is true.

These components may be instantiated by any finite-state abstraction scheme. For instance, if the scheme is predicate abstraction, then $\Omega$ is the set of all valuations to the finite set of predicates with respect to which the predicate abstraction of the program is constructed. For convenience, we treat $\Omega$ as a set of indices instead of abstract contexts. We use $i, j, ...$ to range over $\Omega$ and $\omega_i, \omega_j, ...$ to denote the corresponding abstract contexts indexed by them.

The abstract semantics of the model checker is presented in Figure 1. State $\langle s, \omega \rangle$ is *stuck* if $\nexists a : \langle s, \omega \rangle \hookrightarrow a$. The only kind of state that can get stuck is of the form $\langle \texttt{assert}(e), \omega \rangle$ such that $\omega \notin \delta_e$. State $\langle s, \omega \rangle$ *goes wrong* if $\exists \langle s', \omega' \rangle : (\langle s, \omega \rangle \hookrightarrow^* \langle s', \omega' \rangle$ and $\langle s', \omega' \rangle$ is stuck). Given a program $s$ and an abstract context $\omega$, the model checker determines whether $\langle s, \omega \rangle$ goes wrong. If $\langle s, \omega \rangle$ goes wrong, it reports a counterexample which is a finite trace $\langle s, \omega \rangle \hookrightarrow^*$

$\langle \texttt{assert}(e), \omega' \rangle$ where $\omega' \notin \delta_e$. Otherwise, it returns the finite set of reachable abstract states $\{ a \mid \langle s, \omega \rangle \hookrightarrow^* a \}$ which serves as a proof that the concrete program does not go wrong, provided the model checker is sound. Model checking soundness is typically proved by showing that the abstract semantics simulates the concrete semantics (see for example [29, 25]).

## 3 Type System

Our type system assigns a type of the form $\bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ to each statement in the program, where $A$ and $\forall i \in A : B_i$ are subsets of $\Omega$. Recall that $\Omega$ is finite whence the type is finitary. Intuitively, the type states that it is safe to begin executing the statement in one of contexts $\{ \omega_i \mid i \in A \}$ and, furthermore, if it begins executing in context $\omega_i$ $(i \in A)$ then it will finish executing in one of contexts $\{ \omega_j \mid j \in B_i \}$. Our type system includes the type $\top \triangleq \bigwedge \emptyset$ to handle the case in which $A$ is empty, and the type $\bot \triangleq \bigvee \emptyset$ to handle the case in which any $B_i$ $(i \in A)$ is empty.

The type rules are shown in Figure 2. We say that an abstract state $\langle s, \omega_k \rangle$ is *well-typed* if statement $s$ can be assigned a type that states that it is safe to begin executing $s$ in abstract context $\omega_k$ (see rule (12)).

Rule (13) type checks primitive statement $p$. The type of $p$ captures the effect of the abstract transfer function $\delta_p$ associated with $p$. The side-condition of the rule states that it is safe to begin executing $p$ in any context in $\Omega$ because we have assumed that $\delta_p$ is a total function.

Rule (14) type checks statement $\texttt{assume}(e)$. The side-condition of the rule says that it is safe to begin executing $\texttt{assume}(e)$ in any context in $\Omega$ and, moreover, the first conjunct in its type states that it has the effect of a skip statement if it begins executing in a context in which $e$ is true while the second conjunct in its type states that there does not exist any context in which it finishes executing if it begins executing in a context in which $e$ is false.

Rule (15) type checks statement $\texttt{assert}(e)$. The side-condition of the rule says that it is safe to begin executing $\texttt{assert}(e)$ only in a context in which $e$ is true, and its type states that it has the effect of a skip statement if it begins executing in such a context.

Rule (16) type checks sequentially composed statements. The side-condition says that it is safe to begin executing $s_1$; $s_2$ only in contexts in which it is safe to begin executing $s_1$ and, moreover, if $s_1$ begins executing in such a context, then it must be safe to begin executing $s_2$ in each context in which $s_1$ might finish executing.

Rule (17) type checks branching statements. The side-condition says that it is safe to begin executing $\texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ only in contexts in which it is safe to begin executing both $s_1$ and $s_2$.

Rule (18) type checks looping statements. The side-condition says that it is safe to begin executing $\texttt{while}\ (*)\ \texttt{do}\ s'$ only in contexts in which it is safe to begin executing $s'$ and, moreover, if $s'$ begins executing in such a context, then it must be safe to begin executing $\texttt{while}\ (*)\ \texttt{do}\ s'$ in each context in which

$$\frac{s \;:\; \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i}\omega_j)}{\langle s, \omega_k\rangle \text{ is well-typed}} \quad [k \in A] \tag{12}$$

$$p \;:\; \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in\delta_p(i)}\omega_j) \qquad [A \subseteq \Omega] \tag{13}$$

$$\texttt{assume}(e) \;:\; \bigwedge_{i\in A}(\omega_i \to \omega_i) \;\wedge\; \bigwedge_{i\in B}(\omega_i \to \bot) \qquad [A \subseteq \delta_e \wedge B \subseteq \Omega \setminus \delta_e] \tag{14}$$

$$\texttt{assert}(e) \;:\; \bigwedge_{i\in A}(\omega_i \to \omega_i) \qquad [A \subseteq \delta_e] \tag{15}$$

$$\frac{\begin{array}{c} s_1 \;:\; \bigwedge_{i\in A_1}(\omega_i \to \bigvee_{j\in B_i}\omega_j) \\ s_2 \;:\; \bigwedge_{i\in A_2}(\omega_i \to \bigvee_{j\in B'_i}\omega_j)\end{array}}{s_1; s_2 \;:\; \bigwedge_{i\in A}(\omega_i \to \bigvee_{k\in\bigcup\{\,B'_j \,|\, j\in B_i\,\}}\omega_k)} \quad \left[A \subseteq A_1 \wedge \bigcup_{i\in A} B_i \subseteq A_2\right] \tag{16}$$

$$\frac{\begin{array}{c} s_1 \;:\; \bigwedge_{i\in A_1}(\omega_i \to \bigvee_{j\in B_i}\omega_j) \\ s_2 \;:\; \bigwedge_{i\in A_2}(\omega_i \to \bigvee_{j\in B'_i}\omega_j)\end{array}}{\texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2 \;:\; \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i\cup B'_i}\omega_j)} \quad [A \subseteq A_1 \cap A_2] \tag{17}$$

$$\frac{s' \;:\; \bigwedge_{i\in A'}(\omega_i \to \bigvee_{j\in B_i}\omega_j)}{\texttt{while } (*) \texttt{ do } s' \;:\; \bigwedge_{i\in A}(\omega_i \to \bigvee_{k\in\mu X.(\{i\}\cup\{B_j | j\in X\})}\omega_k)} \quad \left[A \subseteq A' \wedge \bigcup_{i\in A} B_i \subseteq A\right] \tag{18}$$

$\mu X.E$ denotes the least fixed point of function $\lambda X.E : 2^\Omega \to 2^\Omega$

**Fig. 2.** Type Rules

$s'$ might finish executing. Let $\mu X.E$ denote the least fixed point of the function $\lambda X.E : 2^\Omega \to 2^\Omega$. Then, the type of `while` $(*)$ `do` $s'$ states that if the loop begins executing in context $\omega_i$ ($i \in A$), then it will finish executing in one of contexts $\{\,\omega_k \mid k \in \mu X\,.\,(\{i\} \cup \{B_j \,|\, j \in X\})\,\}$, that is: (i) in the base case (0 iterations) the loop will finish executing in the context $\omega_i$ in which it began executing, and (ii) in the inductive case ($n+1$ iterations where $n \geq 0$) the loop will finish executing in one of contexts $\{\,\omega_k \,|\, k \in B_j\,\}$ where $\omega_j$ is a context in which the loop might finish executing in $n$ iterations, in which case in the $n+1^{th}$ iteration, $s'$ will begin executing in context $\omega_j$ and finish executing in one of contexts $\{\,\omega_k \,|\, k \in B_j\,\}$.

## 4 Equivalence

In this section, we prove that a program type checks if and only if the model checker accepts it.

The proof from type checking to model checking is similar to that of type soundness, consisting of Progress (Lemma 1) and Type Preservation (Lemma 2), the key difference being that the step relation is the abstract semantics of the model checker instead of the concrete semantics of the language.

**Lemma 1. (Progress)**
*If $\langle s, \omega_m \rangle$ is well-typed then $\langle s, \omega_m \rangle$ is not stuck.*

*Proof.* See Appendix.

**Lemma 2. (Type Preservation)**
*If $\langle s, \omega_m \rangle$ is well-typed and $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ then $\langle s', \omega_n \rangle$ is well-typed.*

*Proof.* See Appendix.

It is then straightforward to prove that if a program type checks then the model checker accepts it.

**Lemma 3. (From Type Checking to Model Checking)**
*If $\langle s, \omega_m \rangle$ is well-typed then $\langle s, \omega_m \rangle$ does not go wrong.*

*Proof.* Suppose $\langle s, \omega_m \rangle$ is well-typed. We need to prove that $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ implies $\langle s', \omega_n \rangle$ is not stuck. Suppose $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$. From $\langle s, \omega_m \rangle$ is well-typed and $\langle s, \omega_m \rangle \hookrightarrow^t \langle s', \omega_n \rangle$ and lemma (2), we have $\langle s', \omega_n \rangle$ is well-typed. From $\langle s', \omega_n \rangle$ is well-typed and lemma (1), we have $\langle s', \omega_n \rangle$ is not stuck.

The proof from model checking to type checking is constructive and involves building a type derivation from the model constructed by the model checker. The following definitions show how to construct types from the model.

**Definition 1.** $\mathbb{A}^s = \{\, i \in \Omega \mid \langle s, \omega_i \rangle \text{ does not go wrong} \,\}$

**Definition 2.** *Given statement $s$ and $i \in \Omega$, define $\mathbb{B}^{s,i} \subseteq \Omega$ as follows:*

$$\begin{array}{ll}
\mathbb{B}^{s,i} = \delta_p(i) & \textit{if } s = p \\
\mathbb{B}^{s,i} = \{i\} & \textit{if } s = \texttt{assume}(e) \textit{ or } \texttt{assert}(e) \textit{ and } i \in \delta_e \\
\mathbb{B}^{s,i} = \emptyset & \textit{if } s = \texttt{assume}(e) \textit{ or } \texttt{assert}(e) \textit{ and } i \notin \delta_e \\
\mathbb{B}^{s,i} = \bigcup \{\, \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \,\} & \textit{if } s = s_1;\ s_2 \\
\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i} & \textit{if } s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2 \\
\mathbb{B}^{s,i} = \mu X \,.\, (\{i\} \cup \{\, \mathbb{B}^{s',j} \mid j \in X \,\}) & \textit{if } s = \texttt{while } (*) \texttt{ do } s'
\end{array}$$

The key lemma involves showing that the constructed types yield a valid type derivation. It is proved by induction on the structure of the program.

**Lemma 4. (Typability)** $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

*Proof.* See Appendix.

It is then straightforward to prove that if a program is accepted by the model checker then it type checks.

**Lemma 5. (From Model Checking to Type Checking)**
*If $\langle s, \omega_m \rangle$ does not go wrong then $\langle s, \omega_m \rangle$ is well-typed.*

*Proof.* From lemma (4), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$. From $\langle s, \omega_m \rangle$ does not go wrong and defn. (1), we have $m \in \mathbb{A}^s$. From $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \rightarrow \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$ and $m \in \mathbb{A}^s$ and rule (12), we have $\langle s, \omega_m \rangle$ is well-typed.

Finally, we present our main result which states that a program type checks if and only if the model checker accepts it.

**Theorem 1. (Equivalence)**
*$\langle s, \omega \rangle$ is well-typed if and only if $\langle s, \omega \rangle$ does not go wrong.*

*Proof.* Combine lemma (3) and lemma (5).

## 5 Examples

In this section, we illustrate our equivalence result by means of three examples.
*Example 1.* Consider the following program:

$$s_1 \triangleq \text{lock}_1();\, \text{lock}_2() \quad \text{where} \quad \text{lock}() \triangleq \texttt{assert}(s = \texttt{U});\, s := \texttt{L}$$

where U and L denote the unlocked and locked states, respectively. Suppose the model checker is instantiated with predicate abstraction in which case $\Omega$ is a set of program predicates, say $\{\texttt{s}=\texttt{U}, \texttt{s}=\texttt{L}\}$. It is easy to see that state $\langle s_1, \texttt{s}=\texttt{U} \rangle$ goes wrong in the abstract semantics of Figure 1 and is not well-typed in the type system of Figure 2. As a result, both the model checker and the type system reject it.

Notice that although not every state $\langle s, \omega \rangle$ is well-typed in our type system, every statement $s$ is typable (see lemma (4)). For instance, although $\langle s_1, \texttt{s}=\texttt{U} \rangle$ is not well-typed, $s_1$ has the type $\top$. The following example motivates the need for making every statement typable, namely, the need for the type $\top$.
*Example 2.* Consider the following program:

$$s_2 \triangleq \text{lock}_1();\, \texttt{assume}(\mathit{false});\, \text{lock}_2()$$

Assuming the same predicate abstraction as in the previous example, it is easy to see that state $\langle s_2, \texttt{s}=\texttt{U} \rangle$ does not go wrong in the abstract semantics of Figure 1. This is because lock$_2$() is rendered unreachable from state $\langle s_2, \texttt{s}=\texttt{U} \rangle$ in the abstract semantics by the $\texttt{assume}(\mathit{false})$ statement as a result of which the model checker does not even analyze lock$_2$(). However, the type system must type check all code, including code that is *dead*. In particular, it must assign a type to lock$_2$(). It uses the type $\top$ for this purpose. Then, a type derivation for $s_2$ illustrating that $\langle s_2, \texttt{s}=\texttt{U} \rangle$ is well-typed is as follows:

$$\dfrac{\dfrac{\text{lock}_1() \;:\; \texttt{s}=\texttt{U} \rightarrow \texttt{s}=\texttt{L} \qquad \texttt{assume}(\mathit{false}) \;:\; \texttt{s}=\texttt{L} \rightarrow \bot}{\text{lock}_1();\; \texttt{assume}(\mathit{false}) \;:\; \texttt{s}=\texttt{U} \rightarrow \bot} \qquad \text{lock}_2() \;:\; \top}{s_2 \;:\; \texttt{s}=\texttt{U} \rightarrow \bot}$$

*Example 3.* Consider the following program:

$$s_3 \triangleq \{\, \texttt{while}\ (*)\ \texttt{do}\ \{\, \texttt{assume}(i \neq 2);\, i := i + 1 \,\}\,\};\, \texttt{assume}(i = 2)$$

Suppose the abstraction scheme is predicate abstraction and suppose $\Omega = \{\mathsf{i}{=}0, \mathsf{i}{=}1, \mathsf{i}{=}2\}$. Then, each of states $\langle s_3, \mathsf{i}{=}0\rangle$, $\langle s_3, \mathsf{i}{=}1\rangle$, and $\langle s_3, \mathsf{i}{=}2\rangle$ does not go wrong in our abstract semantics and, likewise, each of them is well-typed in our type system since $s_3$ has type $\mathsf{i}{=}0 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \mathsf{i}{=}2$. For instance, a type derivation for $s_3$ illustrating that state $\langle s_3, \mathsf{i}{=}0\rangle$ is well-typed is as follows:

$$\dfrac{\dfrac{\dfrac{\begin{array}{r}\texttt{assume}(i \neq 2)\ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}0 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}2 \to \bot \\ i := i+1\ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \qquad\qquad\end{array}}{\begin{array}{l}\texttt{assume}(i \neq 2);\ i := i+1\ : \\ \qquad\qquad \mathsf{i}{=}0 \to \mathsf{i}{=}1 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \bot\end{array}}}{\begin{array}{l}\texttt{while}\ (*)\ \texttt{do}\ \{\ \texttt{assume}(i \neq 2);\ i := i+1\ \}\ : \\ \qquad\qquad \mathsf{i}{=}0 \to (\mathsf{i}{=}0 \ \vee\ \mathsf{i}{=}1 \ \vee\ \mathsf{i}{=}2)\end{array}} \qquad \begin{array}{c}\texttt{assume}(i = 2)\ : \\ \mathsf{i}{=}0 \to \bot \ \wedge \\ \mathsf{i}{=}1 \to \bot \ \wedge \\ \mathsf{i}{=}2 \to \mathsf{i}{=}2\end{array}}{s_3\ :\ \mathsf{i}{=}0 \to \mathsf{i}{=}2}$$

Thus, both the model checker and the type system accept each of states $\langle s_3, \mathsf{i}{=}0\rangle$, $\langle s_3, \mathsf{i}{=}1\rangle$, and $\langle s_3, \mathsf{i}{=}2\rangle$.

## 6 Related Work

In recent years, there has been a significant surge of interest in type systems for checking temporal safety properties of imperative programs [47, 13, 17, 23, 26]. For instance, consider program $s_3$ in Example 3 above which has the type $\mathsf{i}{=}0 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}1 \to \mathsf{i}{=}2 \ \wedge\ \mathsf{i}{=}2 \to \mathsf{i}{=}2$ in our type system instantiated with the set of abstract contexts $\Omega = \{\mathsf{i}{=}0, \mathsf{i}{=}1, \mathsf{i}{=}2\}$. In CQual [17], which supports references and therefore has a more specialized type system than ours, $s_3$ would be annotated with a constrained polymorphic type:

$$\begin{array}{l} s_3 : \forall c, c'.\ (\mathit{ref}(l), [l \mapsto \mathit{int}(c)]) \to (\mathit{ref}(l), [l \mapsto \mathit{int}(c')])\ / \\ \qquad\qquad \{(c = 0 \Rightarrow c' = 2), (c = 1 \Rightarrow c' = 2), (c = 2 \Rightarrow c' = 2)\}\end{array}$$

where $\mathit{ref}(l)$ is a singleton reference type, namely, the type of a reference to the location $l$, and $\mathit{int}(c)$ is a singleton integer type, namely, the type of the integer constant $c$. Singleton types are not unusual and have also been used in the type systems of languages such as Xanadu [47] and Vault [13] as well as in the type systems of alias types [45] and refinement types [26].

There is a large body of work on bridging different approaches to static analysis, most notably (i) on relating type systems and control-flow analysis for higher-order functional languages, and (ii) on relating data-flow analysis and model checking for first-order imperative languages.

*Type Systems and Control-Flow Analysis.* The Amadio-Cardelli type system [2] with recursive types and subtyping has been shown to be equivalent to a certain 0-CFA-based safety analysis by Palsberg and O'Keefe [36] and to a certain form of constrained types by Palsberg and Smith [37], thereby unifying three different views of typing. Heintze [20] proves that four restrictions of 0-CFA are equivalent to four type systems parameterized by recursive types and subtyping. Palsberg shows that equality-based 0-CFA is equivalent to a type system with recursive types and an unusual notion of subtyping [34]. Palsberg and Pavlopoulou [33] and Amtoft and Turbak [3] show that a class of finitary polyvariant control-flow analyses is equivalent to a type system with finitary polymorphism in the form of union and intersection types. Mossin [28] presents a sound and complete type-based flow analysis in that it predicts a redex if and only if there exists a reduction sequence such that the redex will be reduced. Mossin's approach uses intersection types annotated with flow information; a related approach to flow analysis has been presented by Banerjee [5].

*Data-Flow Analysis and Model Checking.* Schmidt and Steffen [42, 41, 40] relate data-flow analysis and model checking for first-order imperative languages. They show that the information computed by classical iterative data-flow analyses is the same as that obtained by model checking certain modal mu-calculus formulae on the program's trace-based abstract interpretation (*a.i.*), an operational-semantics-based representation of the program's *a.i.* as a computation tree of traces.

## 7 Conclusions

We have presented a type system that is equivalent to a model checker for verifying temporal safety properties of imperative programs. Our result highlights the essence of the relationship between type systems and model checking, provides a methodology for studying their relative expressiveness, is a step towards sharing results between them, and motivates synergistic program analyses that can gain the advantages of both approaches without suffering the drawbacks of either.

Two limitations of our current work are that our language lacks features such as higher-order functions, objects, and concurrency, and the type information extracted from the model constructed by our model checker may not be suitable for human reasoning. We intend to explore these issues in the context of specific verification problems. For instance, see [1] for an approach that infers lock types from executions of multithreaded Java programs in the context of verifying race-freedom.

## Acknowledgments

We originally proved our equivalence result in the setting of the deadline analysis problem for the interrupt calculus. That result can be found in the first author's Master's thesis [29]. We thank the many people who suggested that we prove the result in a more conventional setting such as the one in this paper. The proof technique remains essentially the same. We would also like to thank Alex Aiken and Jakob Rehof for useful discussions and the anonymous reviewers for insightful comments. We were supported by a National Science Foundation ITR Award number 0112628.

## References

1. R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *Proceedings of VMCAI'04*, pages 149–160, January 2004.
2. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4):575–631, 1993.
3. Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Proceedings of ESOP'00*, pages 26–40, 2000.
4. Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of POPL'02*, pages 1–3, 2002.
5. Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *Proceedings of ICFP'97*, pages 1–10, 1997.
6. M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM LOPLAS*, 2(1-4):17–30, 1993.
7. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE'03*, pages 385–395, May 2003.
8. Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of POPL'02*, pages 45–57, 2002.
9. Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of ICFP'01*, pages 193–204, 2001.
10. Patrick Cousot. Types as abstract interpretations. In *Proceedings of POPL'97*, pages 316–331, 1997.
11. Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of POPL'00*, pages 12–25, 2000.
12. M. Debbabi, A. Benzakour, and K. Ktari. A synergy between model-checking and type inference for the verification of value-passing higher-order processes. In *Proceedings of AMAST'98*, pages 214–230, 1999.
13. Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of PLDI'01*, pages 59–69, 2001.
14. Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.
15. Cormac Flanagan and Stephen N. Freund. Type inference against races. In *Proceedings of SAS'04*, 2004.
16. Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *Proceedings of TLDI'05*, January 2005.
17. Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In *Proceedings of PLDI'02*, pages 1–12, 2002.
18. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of CAV'97*, pages 72–83, 1997.
19. Christian Haack and Joe B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of ESOP'03*, pages 284–301, 2003.
20. Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95*, pages 189–206, September 1995.
21. T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of CAV'02*, pages 526–538, 2002.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of SPIN'03*, pages 235–239, 2003.
23. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of POPL'02*, pages 331–342, 2002.

24. G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of POPL'86*, pages 44–57, 1986.
25. Di Ma. *Bounding the Stack Size of Interrupt-Driven Programs*. PhD thesis, Purdue University, 2004.
26. Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of ICFP'03*, 2003.
27. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
28. C. Mossin. Exact flow analysis. In *Proceedings of SAS'97*, pages 250–264, 1997.
29. Mayur Naik. A type system equivalent to a model checker. Master's thesis, Purdue University, 2004.
30. Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In *Proceedings of ESOP'05*, 2005. Full version with proofs available at `http://www.cs.stanford.edu/~mhn/pubs/esop05.html`.
31. K. S. Namjoshi. Certifying model checkers. In *Proceedings of CAV'01*, pages 2–12, 2001.
32. K. S. Namjoshi. Lifting temporal proofs through abstractions. In *Proceedings of VMCAI'03*, pages 174–188, 2003.
33. J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001.
34. Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM TOPLAS*, 20(6):1251–1264, 1998.
35. Jens Palsberg and Di Ma. A typed interrupt calculus. In *Proceedings of FTRTFT'02*, pages 291–310, September 2002.
36. Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM TOPLAS*, 17(4):576–599, July 1995.
37. Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM TOPLAS*, 18(5):519–527, September 1996.
38. Doron Peled, Amir Pnueli, and Lenore D. Zuck. From falsification to verification. In *Proceedings of FSTTCS'01*, pages 292–304, 2001.
39. Doron Peled and Lenore D. Zuck. From model checking to a temporal proof. In *Proceedings of SPIN'01*, pages 1–14, 2001.
40. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of SAS'98*, pages 351–380, 1998.
41. David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of POPL'98*, pages 38–48, 1998.
42. Bernhard Steffen. Data flow analysis as model checking. In *Proceedings of TACS'91, Theoretical Aspects of Computer Science*, pages 346–364, 1991.
43. Li Tan and Rance Cleaveland. Evidence-based model checking. In *Proceedings of CAV'02*, pages 455–470, 2002.
44. Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM TOSEM*, 10(1):5–55, 2001.
45. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of TIC'00*, pages 177–206, 2001.
46. Mitchell Wand. Finding the source of type errors. In *Proceedings of POPL'86*, pages 38–43, 1986.
47. Hongwei Xi. Imperative programming with dependent types. In *Proceedings of LICS'00*, pages 375–387, 2000.

## Appendix

**Lemma 6. (Progress)** *If $\langle s, \omega_m\rangle$ is well-typed then $\langle s, \omega_m\rangle$ is not stuck.*

*Proof.* By induction on the structure of $s$. There are 6 cases depending upon the form of $s$. (In cases (1), (2), (5), and (6), we do not use the hypothesis that $\langle s, \omega_m\rangle$ is well-typed.)

1. $s = p$. Immediate from rule (1) and the fact that $\forall i \in \Omega : \delta_p(i) \neq \emptyset$.
2. $s = \texttt{assume}(e)$. Immediate from rules (2) and (3).
3. $s = \texttt{assert}(e)$. From $\langle s, \omega_m\rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and rule (15), we have $A \subseteq \delta_e$. From $m \in A$ and $A \subseteq \delta_e$, we have $m \in \delta_e$. From $m \in \delta_e$ and rule (4), we have $\langle s, \omega_m\rangle \hookrightarrow \omega_m$, whence $\langle s, \omega_m\rangle$ is not stuck.
4. $s = s_1;\ s_2$. From $\langle s, \omega_m\rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and rule (16), we have $s_1 : \bigwedge_{i\in A'}(\omega_i \to \bigvee_{j\in B'_i} \omega_j)$ and $A \subseteq A'$. From $m \in A$ and $A \subseteq A'$, we have $m \in A'$. From $s_1 : \bigwedge_{i\in A'}(\omega_i \to \bigvee_{j\in B'_i} \omega_j)$ and $m \in A'$ and rule (12), we have $\langle s_1, \omega_m\rangle$ is well-typed. From $\langle s_1, \omega_m\rangle$ is well-typed and the induction hypothesis, we have $\langle s_1, \omega_m\rangle$ is not stuck. From $\langle s_1, \omega_m\rangle$ is not stuck, we have $\exists a :\ \langle s_1, \omega_m\rangle \hookrightarrow a$. There are 3 cases depending upon the form of $a$. In each case, we will prove that $\langle s, \omega_m\rangle$ is not stuck.
   - $a = \omega'$. From rule (5), we have $\langle s, \omega_m\rangle \hookrightarrow \langle s_2, \omega'\rangle$.
   - $a = \mathsf{error}$. From rule (6), we have $\langle s, \omega_m\rangle \hookrightarrow \mathsf{error}$.
   - $a = \langle s'_1, \omega'\rangle$. From rule (7), we have $\langle s, \omega_m\rangle \hookrightarrow \langle s'_1; s_2, \omega'\rangle$.
5. $s = \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. Immediate from either of rules (8) and (9).
6. $s = \texttt{while}\ (*)\ \texttt{do}\ s'$. Immediate from either of rules (10) and (11).

**Lemma 7.** *If $s : \bigwedge_{i\in C}(\omega_i \to \bigvee_{j\in D_i} \omega_j)$ and $m \in C$ and $\langle s, \omega_m\rangle \hookrightarrow \langle s', \omega_n\rangle$ then $s' : \bigwedge_{i\in E}(\omega_i \to \bigvee_{j\in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$.*

*Proof.* See technical report [30].

**Lemma 8. (Single-step Type Preservation)** *If $\langle s, \omega_m\rangle$ is well-typed and $\langle s, \omega_m\rangle \hookrightarrow \langle s', \omega_n\rangle$ then $\langle s', \omega_n\rangle$ is well-typed.*

*Proof.* From $\langle s, \omega_m\rangle$ is well-typed and rule (12), we have $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i\in A}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A$ and $\langle s, \omega_m\rangle \hookrightarrow \langle s', \omega_n\rangle$ and lemma (7), we have $s' : \bigwedge_{i\in A'}(\omega_i \to \bigvee_{j\in B'_i} \omega_j)$ and $n \in A'$. From $s' : \bigwedge_{i\in A'}(\omega_i \to \bigvee_{j\in B'_i} \omega_j)$ and $n \in A'$ and rule (12), we have $\langle s', \omega_n\rangle$ is well-typed.

**Lemma 9. (Multi-step Type Preservation)** *If $\langle s, \omega_m\rangle$ is well-typed and $\langle s, \omega_m\rangle \hookrightarrow^t \langle s', \omega_n\rangle$ then $\langle s', \omega_n\rangle$ is well-typed.*

*Proof.* By induction on $t$ (using lemma (8)).

**Lemma 10.** *We have:*

$$\mathbb{A}^s \subseteq \begin{cases} \mathbb{A}^{s_1} & \text{if } s = s_1;\ s_2 \\ \mathbb{A}^{s_1} \cap \mathbb{A}^{s_2} & \text{if } s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2 \\ \mathbb{A}^{s'} & \text{if } s = \texttt{while } (*) \texttt{ do } s' \end{cases}$$

*Proof.* See technical report [30].

**Lemma 11.** *If* $s = s_1;\ s_2$ *then* $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$. *If* $s = \texttt{while } (*) \texttt{ do } s'$ *then* $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$.

*Proof.* See technical report [30].

**Lemma 12. (Typability)** $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

*Proof.* By induction on the structure of $s$. There are 6 cases depending upon the form of $s$:

- $s = p$. From defn. (1) and rule (1), we have $\mathbb{A}^s = \Omega$. From defn. (2), we have $\forall i \in \Omega :\ \mathbb{B}^{s,i} = \delta_p(i)$. From $\mathbb{A}^s = \Omega$ and $\forall i \in \mathbb{A}^s :\ \mathbb{B}^{s,i} = \delta_p(i)$ and rule (13), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{assume}(e)$. From defn. (1) and rules (2) and (3), we have $\mathbb{A}^s = \Omega$. From defn. (2), we have $\forall i \in \delta_e :\ \mathbb{B}^{s,i} = \{i\}$ and $\forall i \notin \delta_e :\ \mathbb{B}^{s,i} = \emptyset$. From $\mathbb{A}^s = \Omega$ and $\forall i \in \delta_e :\ \mathbb{B}^{s,i} = \{i\}$ and $\forall i \notin \delta_e :\ \mathbb{B}^{s,i} = \emptyset$ and rule (14), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{assert}(e)$. From defn. (1) and rule (4), we have $\mathbb{A}^s = \delta_e$. From defn. (2), we have $\forall i \in \delta_e : \mathbb{B}^{s,i} = \{i\}$. From $\mathbb{A}^s = \delta_e$ and $\forall i \in \mathbb{A}^s :\ \mathbb{B}^{s,i} = \{i\}$ and rule (15), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = s_1;\ s_2$. From the induction hypothesis, we have $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$. and $s_2 : \bigwedge_{i \in \mathbb{A}^{s_2}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$. From lemma (11), we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$. From defn. (2), we have $\mathbb{B}^{s,i} = \bigcup \{\, \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \,\}$. From $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^{s_2}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$ and $\mathbb{B}^{s,i} = \bigcup\{\mathbb{B}^{s_2,j} | j \in \mathbb{B}^{s_1,i}\}$ and rule (16), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{if } (*) \texttt{ then } s_1 \texttt{ else } s_2$. From the induction hypothesis, we have $s_1 : \bigwedge_{i \in \mathbb{A}^{s_1}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_2}$. From defn. (2), we have $\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i}$. From $s_1 : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_1,i}} \omega_j)$ and $s_2 : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s_2,i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_2}$ and $\mathbb{B}^{s,i} = \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i}$ and rule (17), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.
- $s = \texttt{while } (*) \texttt{ do } s'$. From the induction hypothesis, we have $s' : \bigwedge_{i \in \mathbb{A}^{s'}}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s',i}} \omega_j)$. From lemma (10), we have $\mathbb{A}^s \subseteq \mathbb{A}^{s'}$. From lemma (11), we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$. From defn. (2), we have $\mathbb{B}^{s,i} = \mu X.(\{i\} \cup \{\mathbb{B}^{s',j} \mid j \in X\})$. From $s' : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s',i}} \omega_j)$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s'}$ and $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$ and $\mathbb{B}^{s,i} = \mu X.(\{i\} \cup \{\mathbb{B}^{s',j} \mid j \in X\})$ and rule (18), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.