A Type System Equivalent to a Model Checker

MAYUR NAIK
Intel Research
and
JENS PALSBERG
UCLA

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 makes 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 a variant of function types to capture flow sensitivity and intersection and union types to capture context sensitivity. Our result sheds light on the relationship between type systems and model checking, provides a methodology for studying their relative expressiveness, is a step towards sharing results between the two approaches, and motivates synergistic program analyses involving interplay between them.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*; *formal methods*; *model checking*

General Terms: Verification

Additional Key Words and Phrases: Model checking, type systems

ACM Reference Format:

Nalik, M. and Palsberg, J. 2008. A type system equivalent to a model checker. ACM Trans. Program. Lang. Syst. 30, 5, Article 29 (August 2008), 24 pages. DOI = 10.1145/1387673.1387678 http://doi.acm.org/10.1145/1387673.1387678

This work was supported by National Science Foundation ITR Award number 0112628.
Authors' addresses: M. Naik, Intel Research, 2150 Shattuck Avenue, Penthouse Suite, Berkeley, CA 94704; email: mayur.naik@intel.com; J. Palsberg, Computer Science Department, UCLA, Los Angeles, CA 90095; email: palsberg@ucla.edu.

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 [Cousot 1997; Cousot and Cousot 2000]. 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. Additionally, type systems are usually flow-insensitive, whereas model checkers are flow-sensitive; type systems are usually path-insensitive, whereas model checkers typically explore only feasible paths. These differences between type systems and model checking have a significant consequence: they make 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.

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 [Wand 1986; Johnson and Walz 1986; Beaven and Stansifer 1993; Duggan and Bent 1996; Tip and Dinesh 2001; Chitil 2001; Haack and Wells 2003; Lerner et al. 2007] and, more recently, for languages with concurrency like Java [Flanagan and Freund 2004; Flanagan et al. 2005].

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 [Peled and Zuck 2001; Namjoshi 2001; Peled et al. 2001; Henzinger et al. 2002; Tan and Cleaveland 2002; Namjoshi 2003].

1.2 Our Result

In this article, 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 natural and may be instantiated with any finite-state abstraction scheme such as predicate abstraction [Graf and Saidi 1997]. The type system, which is also parametric, type checks exactly

those programs that are accepted by the model checker. It uses a variant of 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 a 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 ω ranges over a finite set of abstract contexts Ω and $\delta_s : \Omega \to 2^\Omega$ is the abstract transfer function associated with s. Intuitively, the above set says that if s begins executing in abstract context ω_i then it may finish executing in an abstract context $\omega_j \in \delta_s(\omega_i)$. For example, in model checkers such as SLAM [Ball and Rajamani 2002], BLAST [Henzinger et al. 2003], and MAGIC [Chaki et al. 2003], Ω 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 finite polymorphic type of the form:
$$\bigwedge_{i \in A} \left(\omega_i \to \bigvee_{j \in B_i} \omega_j \right),$$
where A and $\forall i \in A : B_i$ are finite. This is the most complex form of typing in our type system. 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). Note that the symbol $\to$ is not a function-type arrow in the sense of typed functional languages; we explain the meaning of $\to$ later.

(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: it has been successfully applied in two additional settings, namely, stack-size analysis [Ma 2004] and deadline analysis [Naik 2004] for a class of real-time programs called interrupt-driven programs [Palsberg and Ma 2002].

(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 Debbabi et al. [1999] and Chaki et al. [2002], 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* [Milner 1978]. Usually, *going wrong* is formalized as *getting stuck* in the operational semantics. More formally, for a program s, an initial concrete environment σ, and an initial abstract environment ω that abstracts σ, 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 Article

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 show an application of our technique to real-time systems. In Section 7, we discuss related work. Finally, in Section 8, we conclude with a note on future work.

2. MODEL CHECKER

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

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

A statement s is either a primitive statement p (for instance, an assignment statement), 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. Both $\texttt{assume}(e)$ and $\texttt{assert}(e)$ fail if e is false in the current context. Our goal is to check statically whether an `assert` statement can fail while not exploring execution paths that encounter a failing `assume` statement. 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:

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

where $(*)$ denotes nondeterministic choice and $\bar{e}$ denotes the negation of e. For example, the right-hand side of the first equation uses two `assume` statements to ensure that we take the `then` branch only if e evaluates to true, and that we take the `else` branch only if e evaluates to false.

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 Ω.

—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 δ_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 Ω 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 Ω as a set of indices instead of abstract contexts. We use $i, j, \ldots$ to range over Ω and $\omega_i, \omega_j, \ldots$ to denote the corresponding abstract contexts indexed by them.

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

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

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

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

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

$$\frac{\langle s_1, \omega\rangle \hookrightarrow \omega'}{\langle s_1; s_2, \omega\rangle \hookrightarrow \langle s_2, \omega'\rangle} \tag{5}$$

$$\frac{\langle s_1, \omega\rangle \hookrightarrow \mathsf{halt}}{\langle s_1; s_2, \omega\rangle \hookrightarrow \mathsf{halt}} \tag{6}$$

$$\frac{\langle s_1, \omega\rangle \hookrightarrow \langle s_1', \omega'\rangle}{\langle s_1; s_2, \omega\rangle \hookrightarrow \langle s_1'; s_2, \omega'\rangle} \tag{7}$$

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

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

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

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

Fig. 1. Abstract semantics.

Note that our use of the term "abstract context" is broader than the use of the term in static analysis where it is often used to mean a sequence of call sites. If we were to add function calls to our language then we could let our abstract contexts include sequences of call sites.

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 ω, the model checker determines whether $\langle s, \omega\rangle$ goes wrong.

Once a model checker has determined whether a state goes wrong, it can output useful information. If $\langle s, \omega\rangle$ goes wrong, it can report 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 can return the finite set of reachable abstract states $\{a \,|\, \langle s, \omega\rangle \hookrightarrow^* a\}$ 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 ([Ma 2004; Naik 2004]).

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 Ω. Recall that Ω is finite, whence the type is finite. Intuitively, the type states that it is safe to begin executing the statement in one of the contexts $\{\, \omega_i \mid i \in A \,\}$. The type also states that if the statement begins executing in context ω_i ($i \in A$) and eventually terminates, it will be in one of the 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

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

$$p \;:\; \bigwedge_{i\in\Omega}(\omega_i \rightarrow \bigvee_{j\in\delta_p(i)}\omega_j) \tag{13}$$

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

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

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

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

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

$\mu X \subseteq A.E$ denotes the least fixed point of function $\lambda X.E : 2^A \rightarrow 2^A$

Fig. 2. Type rules.

$\bot \triangleq \bigvee \emptyset$ to handle the case in which any B_i ($i \in A$) is empty. Notice that $\top$ is a valid type of a statement while $\bot$ is a piece of a type.

The type rules are shown in Figure 2. We say that a statement s is *typable* if we can assign s a type using rules (13)–(18). 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 ω_k (see rule (12)). Every statement is typable (see Lemma 4.6), while only some abstract states are well-typed.

Rule (13) type checks primitive statement p. The type of p captures the effect of the abstract transfer function δ_p associated with p. Note that the type states that it is safe to begin executing p in any context in Ω because we have assumed that δ_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 Ω and, furthermore, 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, furthermore, 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 `if` $(*)$ `then` s_1 `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 `while` $(*)$ `do` s' only in contexts in which it is safe to begin executing s' and, furthermore, if s' begins executing in such a context, then it must be safe to begin executing `while` $(*)$ `do` s' in each context in which s' might finish executing. While our types are non-recursive, we need a least fixed point to express index sets in the type rules. Suppose $\lambda X.E : 2^A \to 2^A$ denotes a monotone function. Then, we use $\mu X \subseteq A.E$ to denote the least fixed point of $\lambda X.E$. In our application, E is of the form $(\{i\} \cup \bigcup\{B_j \,|\, j \in X\})$, where $i \in A$ and $\bigcup_{j \in A} B_j \subseteq A$. Notice that E is monotone in X. Then, the type of `while` $(*)$ `do` s' states that if the loop begins executing in context ω_i $(i \in A)$, then it might finish executing in one of the contexts $\{\, \omega_k \,|\, k \in \mu X \subseteq A\,.\,(\{i\} \cup \bigcup\{B_j \,|\, j \in X\})\,\}$. Intuitively, suppose the loop begins executing in context ω_i. Then, in the base case (0 iterations), the loop will finish executing in context ω_i. In the inductive case ($n+1$ iterations where $n \geq 0$), suppose ω_j is a context in which the loop finishes executing after n iterations. Then, s' will begin executing in context ω_j and might finish executing in one of contexts $\{\, \omega_k \,|\, k \in B_j \,\}$, and hence the loop might finish executing in one of contexts $\{\, \omega_k \,|\, k \in B_j \,\}$ after $n+1$ iterations.

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 4.1) and Typability Preservation (Lemma 4.2), 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 4.1 (PROGRESS). *If $\langle s, \omega_m \rangle$ is well-typed then $\langle s, \omega_m \rangle$ is not stuck.*

PROOF. See Lemma A.1 in the appendix. □

LEMMA 4.2 (TYPABILITY 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. See Lemma A.5 in the appendix. □

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

LEMMA 4.3 (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 $\langle s, \omega_m \rangle \hookrightarrow^* \langle s', \omega_n \rangle$ implies $\langle s', \omega_n \rangle$ is not stuck. Suppose $\langle s, \omega_m \rangle \hookrightarrow^* \langle s', \omega_n \rangle$. From $\langle s, \omega_m \rangle$ is

well-typed and $\langle s, \omega_m \rangle \hookrightarrow^* \langle s', \omega_n \rangle$ and Lemma 4.2, we have $\langle s', \omega_n \rangle$ is well-typed. From $\langle s', \omega_n \rangle$ is well-typed and Lemma 4.1, 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. Intuitively, Definition 4.4 provides the set of pre-conditions for a given statement, while Definition 4.5 provides the set of post-conditions for a given pre-condition of a given statement.

Definition 4.4. $\mathbb{A}^s = \{\, i \in \Omega \mid \langle s, \omega_i \rangle \text{ does not go wrong} \,\}$

Definition 4.5. Given statement s and $i \in \Omega$, define $\mathbb{B}^{s,i} \subseteq \Omega$ as follows:

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

For each statement s, we will use sets $\mathbb{A}^s$ and $\mathbb{B}^{s,i}$ to construct its type. The key lemma involves showing that the constructed type yields a valid type derivation. We prove the lemma by induction on the structure of the statement.

LEMMA 4.6 (TYPABILITY). $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$.

PROOF. See Lemma A.11 in the appendix. □

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

LEMMA 4.7 (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 4.6, we have $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$. From $\langle s, \omega_m \rangle$ does not go wrong and Defn. 4.4, we have $m \in \mathbb{A}^s$. From $s : \bigwedge_{i \in \mathbb{A}^s} (\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$ 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 4.8 (EQUIVALENCE). *$\langle s, \omega \rangle$ is well-typed if and only if $\langle s, \omega \rangle$ does not go wrong.*

PROOF. Combine Lemma 4.3 and Lemma 4.7. □

5. EXAMPLES

In this section, we illustrate our equivalence result by means of three examples.

Example 1. Consider the following program:

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

where subscripts 1 and 2 are purely to help identify the statements and have no semantic meaning, and U and L denote the unlocked and locked states, respectively. Suppose the model checker is instantiated with an instance of predicate abstraction such that Ω is a set of program predicates, say $\{\mathsf{x{=}U}, \mathsf{x{=}L}\}$. It is easy to see that state $\langle s_1, \mathsf{x{=}U}\rangle$ goes wrong in the abstract semantics of Figure 1. Additionally, $\langle s_1, \mathsf{x{=}U}\rangle$ is not well-typed in the type system of Figure 2 because $\text{lock}_1() : \mathsf{x{=}U} \rightarrow \mathsf{x{=}L}$ and $\text{lock}_2() : \mathsf{x{=}U} \rightarrow \mathsf{x{=}L}$, and so we cannot type check $\text{lock}_1(); \text{lock}_2()$. As a result, both the model checker and the type system reject $\langle s_1, \mathsf{x{=}U}\rangle$.

Notice that although not every state $\langle s, \omega\rangle$ is well-typed in our type system, every statement s is typable (see Lemma 4.6). For instance, although $\langle s_1, \mathsf{x{=}U}\rangle$ is not well-typed, s_1 has the type $\top$. The following example motivates the need for making every statement typable.

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

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

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 $\mathsf{i{=}0} \rightarrow \mathsf{i{=}2} \wedge \mathsf{i{=}1} \rightarrow \mathsf{i{=}2} \wedge \mathsf{i{=}2} \rightarrow \mathsf{i{=}2}$. For instance, a type derivation for s_3 illustrating that state $\langle s_3, \mathsf{i{=}0}\rangle$ is well-typed

is as follows:

$$\frac{\dfrac{\dfrac{\dfrac{\texttt{assume}(i \neq 2) \ : \ \mathsf{i{=}0} \rightarrow \mathsf{i{=}0} \ \wedge \ \mathsf{i{=}1} \rightarrow \mathsf{i{=}1} \ \wedge \ \mathsf{i{=}2} \rightarrow \bot \qquad i := i+1 \ : \ \mathsf{i{=}0} \rightarrow \mathsf{i{=}1} \ \wedge \ \mathsf{i{=}1} \rightarrow \mathsf{i{=}2}}{\texttt{assume}(i \neq 2); \ i := i+1 \ : \ \mathsf{i{=}0} \rightarrow \mathsf{i{=}1} \ \wedge \ \mathsf{i{=}1} \rightarrow \mathsf{i{=}2} \ \wedge \ \mathsf{i{=}2} \rightarrow \bot} \qquad \texttt{assume}(i = 2) \ : \ \mathsf{i{=}0} \rightarrow \bot \wedge \ \mathsf{i{=}1} \rightarrow \bot \wedge \ \mathsf{i{=}2} \rightarrow \mathsf{i{=}2}}{\texttt{while}\ (*)\ \texttt{do}\ \{\ \texttt{assume}(i \neq 2);\ i := i+1\ \} \ : \ \mathsf{i{=}0} \rightarrow (\mathsf{i{=}0} \ \vee \ \mathsf{i{=}1} \ \vee \ \mathsf{i{=}2}) \wedge \ \mathsf{i{=}1} \rightarrow (\mathsf{i{=}1} \vee \mathsf{i{=}2}) \wedge \ \mathsf{i{=}2} \rightarrow \mathsf{i{=}2}}}{s_3 \ : \ \mathsf{i{=}0} \rightarrow \mathsf{i{=}2} \wedge \mathsf{i{=}1} \rightarrow \mathsf{i{=}2} \wedge \mathsf{i{=}2} \rightarrow \mathsf{i{=}2}}$$

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. DEADLINE ANALYSIS

We have applied our technique to a class of real-time programs called interrupt-driven programs that receive and handle interrupts from their environment [Palsberg and Ma 2002]. In particular, we studied the deadline analysis problem which decides whether a given interrupt-driven program will handle every interrupt within its deadline. For some real-time systems such as airbags or implanted medical devices, a deadline violation can be catastrophic. We illustrate the model checker, the type system, and the equivalence result for deadline analysis by means of an example; we refer the interested reader to Naik [2004] for full details.

Consider the following interrupt-driven program:

```
ei0                 handler 1 {          handler 2 {
loop {                  skip1                skip21
    skip0               iret1                ei2
}                   }                        skip22
                                             iret2
                                         }
```

It consists of a main function and two interrupt handlers each of which handles interrupts from a separate interrupt source in the environment. We have labeled each statement uniquely for ease of reference. The program can control interrupt handling in the body of the main function and each interrupt handler by manipulating the interrupt mask register, denoted imr, which consists of a master bit, denoted $imr(0)$, plus one bit per interrupt source, denoted $imr(1)$ and $imr(2)$ for interrupt sources 1 and 2, respectively. At any instant during execution, interrupt source u is *enabled* if $imr(0) = imr(u) = 1$ and *disabled* otherwise.

The program begins execution at the main function with $imr = 011$ (the master bit is 0), enables interrupt handling by executing statement ei which sets the master bit to 1, and loops forever. In this example, we assume that each interrupt source has *period* 40, that is, the minimum time between the arrivals of successive interrupts from each source is 40 units. We also assume that

each interrupt source has *deadline* 40, that is, the time from the arrival of an interrupt from the source to the completion of its handling by the corresponding interrupt handler must be no more than 40 units, or else we have a deadline violation. Finally, we assume that each primitive statement such as ei, skip, and iret takes 5 units of time to execute.

When interrupt handler u is invoked, the interrupt mask register is pushed on the stack and all interrupt handling is disabled by setting $imr(0) = imr(u) = 0$. An interrupt handler can enable interrupt handling by executing statement ei and thereby allow interruptions by other interrupt handlers but not by itself, that is, an interrupt handler cannot be called recursively. An interrupt handler returns by executing iret which also restores the interrupt mask register from the stack. In our example, `handler 1` executes uninterrupted but `handler 2` first performs a critical computation (denoted by statement skip_{21}), then executes ei and thereby allows interruptions, and finally performs a non-critical computation (denoted by statement skip_{22}). Note that in effect, `handler 1` has higher priority: `handler 2` allows `handler 1` to interrupt it as soon as it finishes its critical computation.

We have defined a model checker and a type system that are equivalent, and both guarantee that every interrupt will be handled within its deadline. In our model checker, an abstract state is a 4-tuple $\langle a, imr, \overline{I}, \omega\rangle$ where a is the program counter, imr is the value of the interrupt mask register, $\overline{I}$ is the latency vector (described below), and ω is the stack generated by the grammar $\omega ::= \langle a, imr\rangle :: \omega \mid \mathsf{nil}$. The stack, and hence the reachable state-space, is finite since an interrupt handler cannot be called recursively. In our example, the latency vector $\overline{I}$ is a vector of two integers such that for interrupt source u:

—if $\overline{I}(u) \geq 0$ then an interrupt from source u has been *pending* for $\overline{I}(u)$ time units, in particular, if $\overline{I}(u) = 0$ then an interrupt from source u has just arrived; and

—if $\overline{I}(u) < 0$ then no interrupt from source u has been pending and, additionally, the next interrupt from source u will arrive no earlier than in $|\overline{I}(u)|$ time units.

A fraction of the reachable state-space of this example program explored by our model checker is as follows:

$$\langle \mathsf{ei}_0; \mathsf{loop\ skip}_0, 011, 0\ 0, \mathsf{nil}\rangle \tag{19}$$

$$\hookrightarrow \langle \mathsf{loop\ skip}_0, 111, 5\ 5, \mathsf{nil}\rangle \tag{20}$$

$$\hookrightarrow \langle \mathsf{skip}_{21}; \mathsf{ei}_2; \mathsf{skip}_{22}; \mathsf{iret}_2, 010, 5\ 5, \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{21}$$

$$\hookrightarrow \langle \mathsf{ei}_2; \mathsf{skip}_{22}; \mathsf{iret}_2, 010, 10\ 10, \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{22}$$

$$\hookrightarrow \langle \mathsf{skip}_{22}; \mathsf{iret}_2, 110, 15\ 15, \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{23}$$

$$\hookrightarrow \langle \mathsf{skip}_1; \mathsf{iret}_1, 000, 15\ 15, \langle \mathsf{skip}_{22}; \mathsf{iret}_2, 110\rangle :: \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{24}$$

$$\hookrightarrow \langle \mathsf{iret}_1, 000, 20\ 20, \langle \mathsf{skip}_{22}; \mathsf{iret}_2, 110\rangle :: \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{25}$$

$$\hookrightarrow \langle \mathsf{skip}_{22}; \mathsf{iret}_2, 110, -15\ 25, \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{26}$$

$$\hookrightarrow \langle \mathsf{iret}_2, 110, -10\ 30, \langle \mathsf{loop\ skip}_0, 111\rangle :: \mathsf{nil}\rangle \tag{27}$$

$$\hookrightarrow \langle \mathsf{loop\ skip}_0, 111, -5\ -5, \mathsf{nil}\rangle \tag{28}$$

This abstract execution path starts in abstract state $\langle \mathsf{ei}_0; \mathsf{loop}\ \mathsf{skip}_0, 011, 0\,0, \mathsf{nil}\rangle$. In this state, we are ready to execute the main function, *imr* is set to 011, interrupts from both sources have just arrived, and the stack is empty. The path first executes ei_0 and thereby sets $imr(0)$ to 1; both numbers in the latency vector are incremented by 5 and we arrive at state (20). Both interrupt handlers are now enabled and the semantics arbitrarily picks `handler 2` to be called. The current statement and *imr* are pushed on the stack, we jump to the body of `handler 2`, $imr(0)$ and $imr(2)$ are set to 0, and we arrive at state (21). Notice that we model this operation as being instantaneous; it takes no time. The next two steps execute skip_{21}, which only updates the latency vector, and ei_2, which sets $imr(0)$ to 1 and updates the latency vector, to arrive at state (23). In this state, `handler 1` is enabled while `handler 2` is disabled. Now `handler 1` is called so the current statement and *imr* are pushed on the stack, we jump to the body of `handler 1`, $imr(0)$ and $imr(1)$ are set to 0, and we arrive at state (24). Next, `handler 1` executes skip_1, which only updates the latency vector, and iret_1, which updates the latency vector and returns to the statement and *imr* stored on the stack, arriving at state (26). Notice that the first number in the latency vector in state (26) is negative; we obtain that number by taking the previous number 20, adding the time increment 5, and then subtracting the period 40. Therefore, in state (26) no interrupt from source 1 is pending and the next interrupt from source 1 will arrive in 15 or more time units. Finally, `handler 2` executes skip_{22} and iret_2 and returns to the main function in state (28).

The numbers in the latency vector in each abstract state in the above execution path as well as in the rest of the reachable state-space are always less than the deadlines. The model checker is sound and therefore guarantees that no deadline violation will occur in any execution of the program. Our type system is equivalent to the model checker and therefore guarantees the absence of deadline violations as well. The types of interrupt handlers and statements have the same structure that we used for the types of statements in our WHILE language. The types of the two interrupt handlers in our example program in compact notation are as follows:

$$\text{type of } \texttt{handler 1} = \ldots \wedge (15\ 15 \xrightarrow{110} -15\ 25) \wedge (5\ 20 \xrightarrow{110} -25\ 30) \wedge \ldots$$

$$\text{type of } \texttt{handler 2} = \ldots \wedge (5\ 5 \xrightarrow{111} (-5\ -5 \vee 5\ -15 \vee -20\ -5)) \wedge \ldots$$

In the type for `handler 1`, the first conjunct stems from the above abstract execution path: if we call `handler 1` from state (23) in which the latency vector is [15 15] and the *imr* is 110, then we will return from the handler in state (26) in which the latency vector is [–15 25]. Likewise, in the type for `handler 2`, the displayed conjunct includes the case we saw above where we call `handler 2` in a state with latency vector [5 5] and the *imr* is 111, and we return to a state with latency vector [–5 –5].

7. RELATED WORK

In recent years, there has been a significant surge of interest in type systems for checking temporal safety properties of imperative programs [Xi 2000; DeLine and Fahndrich 2001; Foster et al. 2002; Igarashi and Kobayashi 2002;

Mandelbaum et al. 2003]. For instance, consider program s_3 in Example 3 from Section 5 which has the type $\mathsf{i}{=}0 \rightarrow \mathsf{i}{=}2 \wedge \mathsf{i}{=}1 \rightarrow \mathsf{i}{=}2 \wedge \mathsf{i}{=}2 \rightarrow \mathsf{i}{=}2$ in our type system instantiated with the set of abstract contexts $\Omega = \{\mathsf{i}{=}0, \mathsf{i}{=}1, \mathsf{i}{=}2\}$. In CQual [Foster et al. 2002], which supports references and therefore has a more specialized type system than ours, s_3 would be annotated with a constrained polymorphic type:

$$s_3 : \forall c, c'. \quad (ref(l), [l \mapsto int(c)]) \rightarrow (ref(l), [l \mapsto int(c')]) \; / \; \{(c = 0 \Rightarrow c' = 2), (c = 1 \Rightarrow c' = 2), (c = 2 \Rightarrow c' = 2)\}$$

where $ref(l)$ is a singleton reference type, namely, the type of a reference to the location l, and $int(c)$ 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 [Xi 2000] and Vault [DeLine and Fahndrich 2001] as well as in the type systems of alias types [Walker and Morrisett 2001] and refinement types [Mandelbaum et al. 2003].

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 [Amadio and Cardelli 1993] with recursive types and subtyping has been shown to be equivalent to a certain 0-CFA-based safety analysis by Palsberg and O'Keefe [1995] and to a certain form of constrained types by Palsberg and Smith [1996], thereby unifying three different views of typing. Heintze [1995] proves that four restrictions of 0-CFA are equivalent to four type systems parameterized by recursive types and subtyping. [Palsberg 1998] shows that equality-based 0-CFA is equivalent to a type system with recursive types and an unusual notion of subtyping. Our result is most closely related to the results of Palsberg and Pavlopoulou [2001] and Amtoft and Turbak [2000], who 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. In Palsberg and Pavlopoulou's type system [2001], the type of a function is of the form $\bigvee_{i \in I} \bigwedge_{j \in J} (\sigma_{ij} \rightarrow \sigma'_{ij})$ where $\rightarrow$ is the function-type arrow, and $\sigma_{ij}, \sigma'_{ij}$ are types. An open problem is to unify their result with ours. Mossin [1997] 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 [1997].

Data-flow analysis and model checking. Schmidt and Steffen [Steffen 1991; Schmidt 1998; Schmidt and Steffen 1998] relate dataflow analysis and model checking for first-order imperative languages. They show that the information computed by classical iterative dataflow analyses is the same as that obtained by model checking certain modal mu-calculus formulae on a trace-based abstract interpretation of the program.

8. 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 the two approaches, 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. In particular, types in many widely used type systems tend to be concise, while types in our type systems tend to bulky. We intend to explore these issues in the context of specific verification problems.

Our type system does not have notions of subtyping or dependent types. An open problem is to define a type system with subtyping and/or dependent types which is equivalent to a model checker.

Type systems can handle higher-order functions and dynamic allocation easily, while model checkers traditionally were applied to hardware, which has no such features. An open problem is to define a type system and an equivalent model checker for such features.

APPENDIX

We now give a proof of Lemma 4.1 (here called A.1), and later in the appendix we give proofs of Lemma 4.2 (here called A.5) and Lemma 4.6 (here called A.11).

LEMMA A.1 (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 six 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 $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ 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 $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and $m \in A$. From $s : \bigwedge_{i \in A}(\omega_i \rightarrow \bigvee_{j \in B_i} \omega_j)$ and rule (16), we have $s_1 : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ and $A \subseteq A'$. From $m \in A$ and $A \subseteq A'$, we have $m \in A'$. From $s_1 : \bigwedge_{i \in A'}(\omega_i \rightarrow \bigvee_{j \in B'_i} \omega_j)$ 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 three 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 $\langle s, \omega_m \rangle \hookrightarrow \langle s_2, \omega' \rangle$.
—$a = \mathsf{halt}$. From rule (6), we have $\langle s, \omega_m \rangle \hookrightarrow \mathsf{halt}$.
—$a = \langle s_1', \omega' \rangle$. 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 rules (8) and (9).

(6) $s = \texttt{while}\ (*)\ \texttt{do}\ s'$. Immediate from rules (10) and (11). □

We next prove Lemma 4.2 (here called A.5). We first prove Lemmas A.2–A.4 which are required in the proof.

LEMMA A.2. *If* $s : \bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ *and* $m \in A$ *and* $\langle s, \omega_m \rangle \hookrightarrow \omega_n$ *then* $n \in B_m$.

PROOF. By case analysis of the rule used in $\langle s, \omega_m \rangle \hookrightarrow \omega_n$. There are four cases depending upon which one of rules (1), (2), (4), and (11) is used.

—Rule (1). We have $s = p$ and $n \in \delta_p(m)$. From $p : \bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and rule (13), we have $\forall i \in A : B_i = \delta_p(i)$. From $\forall i \in A : B_i = \delta_p(i)$ and $m \in A$, we have $B_m = \delta_p(m)$. From $n \in \delta_p(m)$ and $B_m = \delta_p(m)$, we have $n \in B_m$.
—Rule (2). We have $s = \texttt{assume}(e)$ and $m \in \delta_e$ and $n = m$. From $\texttt{assume}(e) : \bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and rule (14), we have $\forall i \in A : (i \in \delta_e \Rightarrow B_i = \{i\}) \wedge (i \notin \delta_e \Rightarrow B_i = \emptyset)$. From $\forall i \in A : (i \in \delta_e \Rightarrow B_i = \{i\}) \wedge (i \notin \delta_e \Rightarrow B_i = \emptyset)$ and $m \in A$ and $m \in \delta_e$, we have $m \in B_m$. From $n = m$ and $m \in B_m$, we have $n \in B_m$.
—Rule (4). We have $s = \texttt{assert}(e)$ and $m \in \delta_e$ and $n = m$. From $\texttt{assert}(e) : \bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and rule (15), we have $\forall i \in A : B_i = \{i\}$. From $\forall i \in A : B_i = \{i\}$ and $m \in A$, we have $m \in B_m$. From $n = m$ and $m \in B_m$, we have $n \in B_m$.
—Rule (11). We have $s = \texttt{while}\ (*)\ \texttt{do}\ s'$ and $n = m$. From $\texttt{while}\ (*)\ \texttt{do}\ s' : \bigwedge_{i \in A}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and rule (18), we have $\forall i \in A : i \in B_i$. From $\forall i \in A : i \in B_i$ and $m \in A$, we have $m \in B_m$. From $n = m$ and $m \in B_m$, we have $n \in B_m$. □

LEMMA A.3. *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. By induction on the structure of s. There are five cases depending upon which one of rules (5), (7), (8), (9), and (10) is used in $\langle s, \omega_m \rangle \hookrightarrow \langle s', \omega_n \rangle$.

—Rule (5). We have $s = s_1; s_2$ and $s' = s_2$. From $s_1; s_2 : \bigwedge_{i \in C}(\omega_i \to \bigvee_{j \in D_i} \omega_j)$ and rule (16), we have:

$$\frac{\begin{array}{c} s_1 \ : \ \bigwedge_{i \in A_1}(\omega_i \to \bigvee_{j \in B_i} \omega_j) \\ s_2 \ : \ \bigwedge_{i \in A_2}(\omega_i \to \bigvee_{j \in B_i'} \omega_j) \end{array}}{s_1; s_2 \ : \ \bigwedge_{i \in A}(\omega_i \to \bigvee_{k \in \bigcup\{B_j' \mid j \in B_i\}} \omega_k)} \qquad \left[A \subseteq A_1 \text{ and } \bigcup_{i \in A} B_i \subseteq A_2 \right]$$

where $C = A$ and $\forall i \in C : D_i = \bigcup_{j \in B_i} B_j'$. Choose $E = A_2$ and $\forall i \in E : F_i = B_i'$. We will prove that $s_2 : \bigwedge_{i \in E}(\omega_i \to \bigvee_{j \in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$.

From $s_2 : \bigwedge_{i \in A_2}(\omega_i \to \bigvee_{j \in B_i'} \omega_j)$ and $E = A_2$ and $\forall i \in E : F_i = B_i'$, we have $s_2 : \bigwedge_{i \in E}(\omega_i \to \bigvee_{j \in F_i} \omega_j)$.

From $m \in C$ and $C = A \subseteq A_1$, we have $m \in A$ and $m \in A_1$. From $\langle s_1; s_2, \omega_m\rangle \hookrightarrow \langle s_2, \omega_n\rangle$ and rule (5), we have $\langle s_1, \omega_m\rangle \hookrightarrow \omega_n$. From $s_1 : \bigwedge_{i\in A_1}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A_1$ and $\langle s_1, \omega_m\rangle \hookrightarrow \omega_n$ and Lemma A.2, we have $n \in B_m$. From $m \in A$ and $n \in B_m$ and $\bigcup_{i\in A} B_i \subseteq A_2 = E$, we have $n \in E$.

We have:

$$\begin{aligned} F_n &= B'_n \\ &\subseteq \bigcup_{j\in B_m} B'_j \quad (\text{from } n \in B_m) \\ &= D_m \end{aligned}$$

—Rule (7). We have $s = s_1; s_2$ and $s' = s'_1; s_2$. From $s_1; s_2 : \bigwedge_{i\in C}(\omega_i \to \bigvee_{j\in D_i} \omega_j)$ and rule (16), we have:

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

where $C = A$ and $\forall i \in C : D_i = \bigcup_{j\in B_i} B'_j$. From $m \in C$ and $C = A \subseteq A_1$, we have $m \in A_1$. From $\langle s_1; s_2, \omega_m\rangle \hookrightarrow \langle s'_1; s_2, \omega_n\rangle$ and rule (7), we have $\langle s_1, \omega_m\rangle \hookrightarrow \langle s'_1, \omega_n\rangle$. From $s_1 : \bigwedge_{i\in A_1}(\omega_i \to \bigvee_{j\in B_i} \omega_j)$ and $m \in A_1$ and $\langle s_1, \omega_m\rangle \hookrightarrow \langle s'_1, \omega_n\rangle$ and the induction hypothesis, we have $s'_1 : \bigwedge_{i\in A_3}(\omega_i \to \bigvee_{j\in B''_i} \omega_j)$ and $n \in A_3$ and $B''_n \subseteq B_m$. Choose $E = \{n\}$ and $\forall i \in E : F_i = \bigcup_{j\in B''_i} B'_j$. We will prove that $s'_1;\ s_2 \ : \ \bigwedge_{i\in E}(\omega_i \to \bigvee_{j\in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$. Notice that the choice of $E = \{n\}$ gives $s'_1;\ s_2$ a "monovariant" type, that is, a type with just one conjunct.

From $m \in C$ and $C = A$, we have $m \in A$. From $m \in A$ and $\bigcup_{i\in A} B_i \subseteq A_2$, we have $B_m \subseteq A_2$. From $s'_1 : \bigwedge_{i\in A_3}(\omega_i \to \bigvee_{j\in B''_i} \omega_j)$ and $s_2 : \bigwedge_{i\in A_2}(\omega_i \to \bigvee_{j\in B'_i} \omega_j)$ and $n \in A_3$ and $B''_n \subseteq B_m \subseteq A_2$ and rule (16), we have $s'_1;\ s_2 : \bigwedge_{i\in\{n\}}(\omega_i \to \bigvee_{k\in\bigcup\{B'_j \mid j\in B''_i\}} \omega_k)$. From $s'_1;\ s_2 : \bigwedge_{i\in\{n\}}(\omega_i \to \bigvee_{k\in\bigcup\{B'_j \mid j\in B''_i\}} \omega_k)$ and $E = \{n\}$ and $\forall i \in E :\ F_i = \bigcup_{j\in B''_i} B'_j$, we have $s'_1;\ s_2 \ : \ \bigwedge_{i\in E}(\omega_i \to \bigvee_{j\in F_i} \omega_j)$.

From $E = \{n\}$, we have $n \in E$.

We have:

$$\begin{aligned} F_n &= \bigcup_{j\in B''_n} B'_j \\ &\subseteq \bigcup_{j\in B_m} B'_j \quad (\text{from } B''_n \subseteq B_m) \\ &= D_m \end{aligned}$$

—Rule (8). We have $s = \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$ and $s' = s_1$ and $n = m$. From $\texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \bigwedge_{i\in C}(\omega_i \to \bigvee_{j\in D_i} \omega_j)$ and rule (17), we have:

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

where $C = A$ and $\forall i \in C : D_i = B_i \cup B'_i$. Choose $E = A_1$ and $\forall i \in E : F_i = B_i$. We will prove that $s_1 : \bigwedge_{i\in E}(\omega_i \to \bigvee_{j\in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$.

From $s_1 : \bigwedge_{i \in A_1}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and $E = A_1$ and $\forall i \in E : F_i = B_i$, we have $s_1 : \bigwedge_{i \in E}(\omega_i \to \bigvee_{j \in F_i} \omega_j)$.

From $n = m$ and $m \in C$ and $C = A \subseteq A_1 = E$, we have $n \in E$.

We have $F_n = B_n = B_m \subseteq B_m \cup B'_m = D_m$.

—Rule (9). The proof is similar to that of the preceding item.

—Rule (10). We have $s = \texttt{while}\ (*)\ \texttt{do}\ s_1$ and $s' = s_1;\ \texttt{while}\ (*)\ \texttt{do}\ s_1$ and $n = m$. From $\texttt{while}\ (*)\ \texttt{do}\ s_1 : \bigwedge_{i \in C}(\omega_i \to \bigvee_{j \in D_i} \omega_j)$ and rule (18), we have:

$$\frac{s_1 \ : \ \bigwedge_{i \in A'}(\omega_i \to \bigvee_{j \in B_i} \omega_j)}{\texttt{while}\ (*)\ \texttt{do}\ s_1 \ : \ \bigwedge_{i \in A}(\omega_i \to \bigvee_{k \in \mu X \subseteq A.(\{i\} \cup \bigcup\{B_j \mid j \in X\})} \omega_k)} \left[\bigcup_{i \in A} B_i \subseteq A \subseteq A'\right]$$

where $C = A$ and $\forall i \in C : D_i = \mu X \subseteq A.(\{\, i \,\} \cup \bigcup\{\, B_j \mid j \in X \,\})$. Choose $E = A$ and $\forall i \in E : F_i = \bigcup_{j \in B_i} \mu X \subseteq A.(\{\, j \,\} \cup \bigcup\{\, B_k \mid k \in X \,\})$. We will prove that $s_1;\ \texttt{while}\ (*)\ \texttt{do}\ s_1 : \bigwedge_{i \in E}(\omega_i \to \bigvee_{j \in F_i} \omega_j)$ and $n \in E$ and $F_n \subseteq D_m$.

From $s_1 : \bigwedge_{i \in A'}(\omega_i \to \bigvee_{j \in B_i} \omega_j)$ and
$\texttt{while}\ (*)\ \texttt{do}\ s_1 \ : \ \bigwedge_{i \in A}(\omega_i \to \bigvee_{k \in \mu X \subseteq A.(\{i\} \cup \bigcup\{B_j \mid j \in X\})} \omega_k)$ and $A \subseteq A'$ and $\bigcup_{i \in A} B_i \subseteq A$ and rule (16), we have
$s_1;\ \texttt{while}\ (*)\ \texttt{do}\ s_1 : \bigwedge_{i \in A}(\omega_i \to \bigvee_{l \in \bigcup\{\mu X \subseteq A.(\{j\} \cup \bigcup\{B_k \mid k \in X\}) \mid j \in B_i\}} \omega_l)$ which combined with $E = A$ and $\forall i \in E : F_i = \bigcup_{j \in B_i} \mu X \subseteq A.\ (\{\, j \,\} \cup \bigcup\{\, B_k \mid k \in X \,\})$ proves that $s_1;\ \texttt{while}\ (*)\ \texttt{do}\ s_1 : \bigwedge_{i \in E}(\omega_i \to \bigvee_{j \in F_i} \omega_j)$.

From $n = m$ and $m \in A$ and $E = A$, we have $n \in E$.

We have:

$$\begin{aligned}
F_n &= \bigcup_{j \in B_n} \mu X \subseteq A.(\{\, j \,\} \cup \bigcup\{\, B_k \mid k \in X \,\}) \\
&= \mu X \subseteq A.(B_n \cup \bigcup\{\, B_k \mid k \in X \,\}) \\
&\subseteq \mu X \subseteq A.(\{\, n \,\} \cup B_n \cup \bigcup\{\, B_k \mid k \in X \,\}) \\
&= \mu X \subseteq A.(\{\, n \,\} \cup \bigcup\{\, B_j \mid j \in X \,\}) \\
&= \mu X \subseteq A.(\{m\} \cup \bigcup\{\, B_j \mid j \in X \,\}) \quad (\text{from } n = m) \\
&= D_m \quad \square
\end{aligned}$$

LEMMA A.4 (SINGLE-STEP TYPABILITY 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 A.3, 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. $\square$

LEMMA A.5 (MULTISTEP TYPABILITY 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. We will prove the stronger statement that for every natural number t, $t \geq 0$, 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. The proof proceeds by induction on t (using Lemma A.4); we omit the details. □

Finally, we prove Lemma 4.6 (here called A.11). We first prove Lemmas A.6–A.10 which are required in the proof.

LEMMA A.6. *If* $\langle s, \omega\rangle \hookrightarrow^* \omega'$ *then* $\langle s; s', \omega\rangle \hookrightarrow^* \langle s', \omega'\rangle$.

PROOF. We will prove the stronger statement that for every natural number $t \geq 1$, if $\langle s, \omega\rangle \hookrightarrow^t \omega'$ then $\langle s; s', \omega\rangle \hookrightarrow^t \langle s', \omega'\rangle$. We proceed by induction on t. In the base case of $t = 1$, the result is immediate from rule (5). In the induction step, suppose $\langle s, \omega\rangle \hookrightarrow^{t+1} \omega'$. We must have $\langle s, \omega\rangle \hookrightarrow \langle s'', \omega''\rangle \hookrightarrow^t \omega'$. From $\langle s, \omega\rangle \hookrightarrow \langle s'', \omega''\rangle$ and rule (7), we have $\langle s; s', \omega\rangle \hookrightarrow \langle s''; s', \omega''\rangle$. From $\langle s'', \omega''\rangle \hookrightarrow^t \omega'$ and the induction hypothesis, we have $\langle s''; s', \omega''\rangle \hookrightarrow^t \langle s'; \omega'\rangle$. From $\langle s; s', \omega\rangle \hookrightarrow \langle s''; s', \omega''\rangle$ and $\langle s''; s', \omega''\rangle \hookrightarrow^t \langle s'; \omega'\rangle$, we have $\langle s; s', \omega\rangle \hookrightarrow^{t+1} \langle s', \omega'\rangle$, as required. □

LEMMA A.7. *If* $\langle s, \omega\rangle \hookrightarrow^* \langle s', \omega'\rangle$ *then* $\langle s; s_2, \omega\rangle \hookrightarrow^* \langle s'; s_2, \omega'\rangle$.

PROOF. We will prove the stronger statement that for every natural number $t \geq 1$, if $\langle s, \omega\rangle \hookrightarrow^t \langle s', \omega'\rangle$ then $\langle s; s_2, \omega\rangle \hookrightarrow^t \langle s'; s_2, \omega'\rangle$. We proceed by induction on t. In the base case of $t = 1$, the result is immediate from rule (7). In the induction step, suppose $\langle s, \omega\rangle \hookrightarrow^{t+1} \langle s', \omega'\rangle$. We must have $\langle s, \omega\rangle \hookrightarrow \langle s'', \omega''\rangle \hookrightarrow^t \langle s', \omega'\rangle$. From $\langle s, \omega\rangle \hookrightarrow \langle s'', \omega''\rangle$ and rule (7), we have $\langle s; s_2, \omega\rangle \hookrightarrow \langle s''; s_2, \omega''\rangle$. From $\langle s'', \omega''\rangle \hookrightarrow^t \langle s', \omega'\rangle$ and the induction hypothesis, we have $\langle s''; s_2, \omega''\rangle \hookrightarrow^t \langle s'; s_2, \omega'\rangle$. From $\langle s; s_2, \omega\rangle \hookrightarrow \langle s''; s_2, \omega''\rangle$ and $\langle s''; s_2, \omega''\rangle \hookrightarrow^t \langle s'; s_2, \omega'\rangle$, we have $\langle s; s_2, \omega\rangle \hookrightarrow^{t+1} \langle s'; s_2, \omega'\rangle$, as required. □

LEMMA A.8. *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. Let us first show $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$, if $s = s_1;\ s_2$. Suppose $i \in \mathbb{A}^s$, that is, $\langle s, \omega_i\rangle$ does not go wrong. For the purpose of deriving a contradiction, assume $i \notin \mathbb{A}^{s_1}$. From $i \notin \mathbb{A}^{s_1}$ we have that $\langle s_1, \omega_i\rangle$ goes wrong, that is, we have $\langle s', \omega'\rangle$ such that $\langle s_1, \omega_i\rangle \hookrightarrow^* \langle s', \omega'\rangle$ and $\langle s', \omega'\rangle$ is stuck. From $\langle s_1, \omega_i\rangle \hookrightarrow^* \langle s', \omega'\rangle$ and Lemma A.7 we have $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \langle s'; s_2, \omega'\rangle$. We have that $\langle s'; s_2, \omega'\rangle$ is stuck, contradicting $i \in \mathbb{A}^s$.

The other two cases can be proved in a similar manner; we omit the details. □

LEMMA A.9. *If* $j \in \mathbb{B}^{s,i}$ *then* $\langle s, \omega_i\rangle \hookrightarrow^* \omega_j$.

PROOF. By induction on the structure of s. There are six cases depending upon the form of s:

—$s = p$. From $j \in \mathbb{B}^{s,i}$ and Defn. 4.5, we have $j \in \delta_p(i)$. From $j \in \delta_p(i)$ and rule (1), we have $\langle s, \omega_i\rangle \hookrightarrow \omega_j$.

—$s = \texttt{assume}(e)$. From $j \in \mathbb{B}^{s,i}$ and Defn. 4.5, we have $j = i$ and $i \in \delta_e$. From $j = i$ and $i \in \delta_e$ and rule (2), we have $\langle s, \omega_i\rangle \hookrightarrow \omega_j$.

—$s = \texttt{assert}(e)$. The proof is similar to that of the preceding item.

—$s = s_1;\ s_2$. From $j \in \mathbb{B}^{s,i}$ and Defn. 4.5, we have $j \in \mathbb{B}^{s_2,k}$ for some $k \in \mathbb{B}^{s_1,i}$. From $k \in \mathbb{B}^{s_1,i}$ and the induction hypothesis, we have $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_k$. From $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_k$ and Lemma A.6, we have $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \langle s_2, \omega_k\rangle$. From $j \in \mathbb{B}^{s_2,k}$ and the induction hypothesis, we have $\langle s_2, \omega_k\rangle \hookrightarrow^* \omega_j$. From $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \langle s_2, \omega_k\rangle$ and $\langle s_2, \omega_k\rangle \hookrightarrow^* \omega_j$, we have $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \omega_j$.

—$s = \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. From $j \in \mathbb{B}^{s,i}$ and Defn. 4.5, we have $j \in \mathbb{B}^{s_1,i} \cup \mathbb{B}^{s_2,i}$. Suppose $j \in \mathbb{B}^{s_1,i}$. (The proof of the case in which $j \in \mathbb{B}^{s_2,i}$ is similar.) From rule (8), we have $\langle s, \omega_i\rangle \hookrightarrow \langle s_1, \omega_i\rangle$. From $j \in \mathbb{B}^{s_1,i}$ and the induction hypothesis, we have $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_j$. From $\langle s, \omega_i\rangle \hookrightarrow \langle s_1, \omega_i\rangle$ and $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_j$, we have $\langle s, \omega_i\rangle \hookrightarrow^* \omega_j$.

—$s = \texttt{while}\ (*)\ \texttt{do}\ s'$. From Defn. 4.5, we have $\mathbb{B}^{s,i} = \bigcup_{t\geq 0} B_t$ where:

$$B_0 = \{i\}$$
$$B_t = \bigcup\{\mathbb{B}^{s',k} \mid k \in B_{t-1}\}, t > 0$$

We will first prove that $\forall t : (l \in B_t \Rightarrow \langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_l\rangle)$. The proof is by induction on t. The base case ($t = 0$) is trivial. To prove the induction step, suppose $l \in B_t, t > 0$. From the definition of B_t above, we have $l \in \mathbb{B}^{s',k}$ for some $k \in B_{t-1}$. From $k \in B_{t-1}$ and the induction hypothesis of the induction on t, we have $\langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_k\rangle$. From rule (10), we have $\langle s, \omega_k\rangle \hookrightarrow \langle s'; s, \omega_k\rangle$. From $l \in \mathbb{B}^{s',k}$ and the induction hypothesis of the induction on s, we have $\langle s', \omega_k\rangle \hookrightarrow^* \omega_l$. From $\langle s', \omega_k\rangle \hookrightarrow^* \omega_l$ and Lemma A.6, we have $\langle s'; s, \omega_k\rangle \hookrightarrow^* \langle s, \omega_l\rangle$. From $\langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_k\rangle$ and $\langle s, \omega_k\rangle \hookrightarrow \langle s'; s, \omega_k\rangle$ and $\langle s'; s, \omega_k\rangle \hookrightarrow^* \langle s, \omega_l\rangle$, we have $\langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_l\rangle$, which completes the proof.
From $j \in \mathbb{B}^{s,i}$ and $\mathbb{B}^{s,i} = \bigcup_{t\geq 0} B_t$, we have $j \in B_{t'}$ for some t'. From $j \in B_{t'}$ and $\forall t : (l \in B_t \Rightarrow \langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_l\rangle)$, we have $\langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_j\rangle$. From rule (11), we have $\langle s, \omega_j\rangle \hookrightarrow \omega_j$. From $\langle s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_j\rangle$ and $\langle s, \omega_j\rangle \hookrightarrow \omega_j$, we have $\langle s, \omega_i\rangle \hookrightarrow^* \omega_j$. □

LEMMA A.10. *We have:*

(1) If $s = s_1;\ s_2$ *then* $\bigcup_{i\in\mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$.

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

PROOF. Consider any $i \in \mathbb{A}^s$. From $i \in \mathbb{A}^s$ and Defn. 4.4, we have $\langle s, \omega_i\rangle$ does not go wrong.

—$s = s_1;\ s_2$
Consider any $j \in \mathbb{B}^{s_1,i}$. We need to prove that $j \in \mathbb{A}^{s_2}$. From $j \in \mathbb{B}^{s_1,i}$ and Lemma A.9, we have $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_j$. From $\langle s_1, \omega_i\rangle \hookrightarrow^* \omega_j$ and Lemma A.6, we have $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \langle s_2, \omega_j\rangle$. From $\langle s_1; s_2, \omega_i\rangle$ does not go wrong and $\langle s_1; s_2, \omega_i\rangle \hookrightarrow^* \langle s_2, \omega_j\rangle$, we have $\langle s_2, \omega_j\rangle$ does not go wrong. From $\langle s_2, \omega_j\rangle$ does not go wrong and Defn. 4.4, we have $j \in \mathbb{A}^{s_2}$.

—$s = \texttt{while}\ (*)\ \texttt{do}\ s'$
Consider any $j \in \mathbb{B}^{s',i}$. We need to prove that $j \in \mathbb{A}^s$. From rule (10), we have $\langle s, \omega_i\rangle \hookrightarrow \langle s'; s, \omega_i\rangle$. From $j \in \mathbb{B}^{s',i}$ and Lemma A.9, we have $\langle s', \omega_i\rangle \hookrightarrow^* \omega_j$. From $\langle s', \omega_i\rangle \hookrightarrow^* \omega_j$ and Lemma A.6, we have $\langle s'; s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_j\rangle$. From $\langle s, \omega_i\rangle$ does not go wrong and $\langle s, \omega_i\rangle \hookrightarrow \langle s'; s, \omega_i\rangle \hookrightarrow^* \langle s, \omega_j\rangle$, we have $\langle s, \omega_j\rangle$

does not go wrong. From $\langle s, \omega_j \rangle$ does not go wrong and Defn. 4.4, we have $j \in \mathbb{A}^s$. □

LEMMA A.11 (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 six cases depending upon the form of s:

—$s = p$. From Defn. 4.4 and rule (1), we have $\mathbb{A}^s = \Omega$. From Defn. 4.5, 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. 4.4 and rules (2) and (3), we have $\mathbb{A}^s = \Omega$. From Defn. 4.5, 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. 4.4 and rule (4), we have $\mathbb{A}^s = \delta_e$. From Defn. 4.5, 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 A.8, we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$. From Lemma A.10, we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s_1,i} \subseteq \mathbb{A}^{s_2}$. From Defn. 4.5, 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 $\mathbb{B}^{s,i} = \bigcup \{ \mathbb{B}^{s_2,j} \mid j \in \mathbb{B}^{s_1,i} \}$ 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 A.8, we have $\mathbb{A}^s \subseteq \mathbb{A}^{s_1}$ and $\mathbb{A}^s \subseteq \mathbb{A}^{s_2}$. From Defn. 4.5, 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 A.8, we have $\mathbb{A}^s \subseteq \mathbb{A}^{s'}$. From Lemma A.10, we have $\bigcup_{i \in \mathbb{A}^s} \mathbb{B}^{s',i} \subseteq \mathbb{A}^s$. From Defn. 4.5, we have $\mathbb{B}^{s,i} = \mu X \subseteq \Omega.(\{i\} \cup \bigcup \{\mathbb{B}^{s',j} \mid j \in X\})$. 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 $\mathbb{B}^{s,i} = \mu X \subseteq \Omega.(\{i\} \cup \bigcup \{\mathbb{B}^{s',j} \mid j \in X\})$ and rule (18), we have $s : \bigwedge_{i \in \mathbb{A}^s}(\omega_i \to \bigvee_{j \in \mathbb{B}^{s,i}} \omega_j)$. □

ACKNOWLEDGMENTS

We originally proved our equivalence result in the setting of the deadline analysis problem for interrupt-driven programs. That result can be found in the first author's master's thesis [Naik 2004]. 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. An earlier version of this paper appeared in the 14^{th} European Symposium on Programming (ESOP'05); this revised version contains more detailed explanations, full proofs,

and a discussion of deadline analysis. We thank the anonymous ESOP'05 and TOPLAS reviewers for insightful comments.

REFERENCES

AMADIO, R. M. AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4, 575–631.

AMTOFT, T. AND TURBAK, F. 2000. Faithful translations between polyvariant flows and polymorphic types. In *Proceedings of the 14th European Symposium on Programming*. Springer, 26–40.

BALL, T. AND RAJAMANI, S. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1–3.

BANERJEE, A. 1997. A modular, polyvariant and type-based closure analysis. In *Proceedings of the 2nd ACM SIGPLAN International Conf. on Functional Programming*. ACM Press, 1–10.

BEAVEN, M. AND STANSIFER, R. 1993. Explaining type errors in polymorphic languages. *ACM Lett. on Program. Lang. Syst. 2,* 1-4, 17–30.

BRYLOW, D. AND PALSBERG, J. 2004. Deadline analysis of interrupt-driven software. *IEEE Trans. Soft. Engin. 30,* 10 634–655.

CHAKI, S., CLARKE, E. M., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in C. In *Proceedings of the 25th International on Software Engineering*. IEEE Computer Society Press, 385–395.

CHAKI, S., RAJAMANI, S. K., AND REHOF, J. 2002. Types as models: Model checking message-passing programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 45–57.

CHATTERJEE, K., MA, D., MAJUMDAR, R., ZHAO, T., HENZINGER, T. A., AND PALSBERG, J. 2004. Stack size analysis of interrupt driven software. *Inform. Comput. 194,* 2, 144–174.

CHITIL, O. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*. 193–204.

COUSOT, P. 1997. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 316–331.

COUSOT, P. AND COUSOT, R. 2000. Temporal abstract interpretation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 12–25.

DEBBABI, M., BENZAKOUR, A., AND KTARI, B. 1999. A synergy between model-checking and type inference for the verification of value-passing higher-order processes. In *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*. Springer, 214–230.

DELINE, R. AND FAHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 59–69.

DUGGAN, D. AND BENT, F. 1996. Explaining type inference. *Sci. Comput. Program. 27,* 1, 37–83.

FLANAGAN, C. AND FREUND, S. N. 2004. Type inference against races. *Sci. Comput. Program. 64,* 1, 140–165.

FLANAGAN, C., FREUND, S. N., AND LIFSHIN, M. 2005. Type inference for atomicity. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM Press, 47–58.

FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1–12.

GRAF, S. AND SAIDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer-Aided Verification*. Springer, 72–83.

HAACK, C. AND WELLS, J. B. 2003. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*. Springer, 284–301.

HEINTZE, N. 1995. Control-flow analysis and type systems. In *Proceedings of the 2nd International Symposium on Static Analysis*. Springer, 189–206.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., NECULA, G. C., SUTRE, G., AND WEIMER, W. 2002. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification*. Springer, 526–538.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2003. Software verification with Blast. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software*. Springer, 235–239.

IGARASHI, A. AND KOBAYASHI, N. 2002. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 331–342.

JOHNSON, G. F. AND WALZ, J. A. 1986. A maximium flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM Press, 44–57.

LERNER, B., FLOWER, M., GROSSMAN, D., AND CHAMBERS, C. 2007. Searching for type-error messages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 425–434.

MA, D. 2004. Bounding the stack size of interrupt-driven programs. Ph.D. thesis, Purdue University.

MANDELBAUM, Y., WALKER, D., AND HARPER, R. 2003. An effective theory of type refinements. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 213–225.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MOSSIN, C. 1997. Exact flow analysis. In *Proceedings of the 4th International Symposium on Static Analysis*. Springer, 250–264.

NAIK, M. 2004. A type system equivalent to a model checker. M.S. thesis, Purdue University.

NAMJOSHI, K. S. 2001. Certifying model checkers. In *Proceedings of the 13th International Conference on Computer-Aided Verification*. Springer, 2–12.

NAMJOSHI, K. S. 2003. Lifting temporal proofs through abstractions. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 174–188.

PALSBERG, J. 1998. Equality-based flow analysis versus recursive types. *ACM Trans. Program. Lang. Syst. 20,* 6, 1251–1264.

PALSBERG, J. AND MA, D. 2002. A typed interrupt calculus. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*. Springer, 291–310.

PALSBERG, J. AND O'KEEFE, P. M. 1995. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst. 17,* 4, 576–599.

PALSBERG, J. AND PAVLOPOULOU, C. 2001. From polyvariant flow information to intersection and union types. *J. Funct. Program. 11,* 3, 263–317.

PALSBERG, J. AND SMITH, S. 1996. Constrained types and their expressiveness. *ACM Transactions Program. Lang. Syst. 18,* 5, 519–527.

PELED, D., PNUELI, A., AND ZUCK, L. D. 2001. From falsification to verification. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 292–304.

PELED, D. AND ZUCK, L. D. 2001. From model checking to a temporal proof. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software*. Springer, 1–14.

SCHMIDT, D. AND STEFFEN, B. 1998. Program analysis as model checking of abstract interpretations. In *Proceedings of the 5th International Symposium on Static Analysis*. Springer, 351–380.

SCHMIDT, D. A. 1998. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 38–48.

STEFFEN, B. 1991. Data flow analysis as model checking. In *Proceedings of Theoretical Aspects of Computer Science*. Springer, 346–364.

TAN, L. AND CLEAVELAND, R. 2002. Evidence-based model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification*. Springer, 455–470.

TIP, F. AND DINESH, T. B. 2001. A slicing-based approach for locating type errors. *ACM Trans. Soft. Engin. Method. 10,* 1 , 5–55.

WALKER, D. AND MORRISETT, G. 2001. Alias types for recursive data structures. In *Proceedings of the 3rd International Workshop on Types in Compilation*. Springer, 177–206.

WAND, M. 1986. Finding the source of type errors. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM Press, 38–43.

XI, H. 2000. Imperative programming with dependent types. In *Proceedings of the 15th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 375–387.

Received November 2005; revised September 2007; accepted December 2007