In-place Refinement for Effect Checking

Viktor Kuncak
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, MA 02139, USA
vkuncak@lcs.mit.edu

K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
leino@microsoft.com

Abstract. The refinement calculus is a powerful framework for reasoning about programs, specifications, and refinement relations between programs and specifications.
In this paper we introduce a new refinement calculus construct, *in-place refinement*. We use in-place refinement to prove the correctness of a technique for checking the refinement relation between programs and specifications. The technique is applicable whenever the specification is an idempotent predicate transformer, as is the case for most procedure effects.
In-place refinement is a predicate on the current program state. A command in-place refines a specification in a given state if the effect of every execution of the command in the state is no worse then the effect of some execution of the specification in the state.
We demonstrate the usefulness of the in-place refinement construct by showing the correctness of a precise technique for checking effects of commands in a computer program. The technique is precise because it takes into account the set of possible states in which each command can execute, using the information about the control-flow and expressions of conditional commands. This precision is particularly important for handling aliasing in object-oriented programs that manipulate dynamically allocated data structures.
We have implemented the technique as a part of a side-effect checker for the programming language C#.

1 Introduction

In this paper we consider an instance of the fundamental problem of showing that a computer program respects a specification. This problem is difficult because

* To Appear in AVIS 2003. Version of February 22, 2003, 2:18pm

both the actual behavior of the program and the desirable program behaviors typically correspond to an infinite state transition system (or a system that is for all practical purposes infinite). For example, in an object-oriented program there is practically no upper bound on the number of objects that the program may create. A general way to address the problem of infinitely many states is to work with the finite *descriptions* of programs and specifications.

The focus of this paper is a technique for checking that a program conforms to its specification where the specification has the form of an *effect*. We have applied our technique to the checking of effects called *modifies-clauses* [15]. A modifies-clause is a specification that requires the program to preserve the values in memory locations outside some given region of program state. The presence of dynamically allocated data structures implies that the region of state that the program may access, as well as the region of state specified in a modifies-clause, are not bounded at compile time. Our analysis technique therefore works directly with the description of the program in its *guarded command form* [16], and uses the concept of a *data group* [15] to represent sets of locations of unknown size.

The main result of this paper is the correctness proof of our technique for checking side effects. Our correctness proof applies whenever the specification satisfies simple algebraic properties: idempotence and refinement by an empty command. We therefore hope that our argument provides a general foundation for checking program effects. To show the correctness of our technique, we give a formal semantics to programs and specifications, using the *refinement calculus*.

The refinement calculus [2] is a framework based on weakest precondition calculus [8]. It allows expressing both programs and specifications in a unified notation with precise semantics based on higher-order logic. Sets of states are modeled in the refinement calculus as *predicates*, here denoted by Pred. Programs and specifications are modeled as *predicate transformers*, here denoted by Trans. A predicate transformer is a function from predicates to predicates, whose informal meaning is the following. Consider a command C modeled by a predicate transformer t. If a predicate P denotes a set of states after the command C, then the predicate $t(P)$ is the *weakest precondition* of C with respect to P. $t(P)$ denotes the largest set of states S such that every execution of C from a state $st \in S$ leads to one of the states denoted by P.

The refinement calculus is an expressive framework that naturally models a range of programming language constructs. When used informally, the refinement calculus permits a full range of mathematical techniques for reasoning about programs. Nevertheless, it is possible to build automated tools that check refinement relations for predicate transformers of certain forms, without requiring any interactive theorem proving. In this paper we focus on specifications that are expressed as effects. We have implemented a modifies-clause checker based on this technique, making use of a theorem prover tailored for program checking [7, 10].

Side effects are a simple yet important class of program specifications. The precise information about side effects enables a program checker to prove that properties are preserved across procedure calls, which makes side effect checking an important aspect of tools that increase software reliability. Side effects can be specified at different levels of precision and conciseness, starting from the most conservative approximation. This property makes side effects a good candidate for lightweight specifications. Our initial experience with the modifies-clause checker for C# as well as the previous experience with ESC/Modula-3 [7] suggest that the annotation burden of specifying side effects is acceptable.

To show the correctness of the effect checking technique in [15], we introduce a new construct of the refinement calculus, *in-place refinement*. In-place refinement is an operation $\sqsubseteq^I$ that takes two predicate transformers t_1 and t_2 and returns a predicate:

$$\sqsubseteq^I \ : \ \mathsf{Trans} \to \mathsf{Trans} \to \mathsf{Pred}$$

$$(t_1 \sqsubseteq^I t_2) \ st = \forall P. \ (t_1 \ P \ st) \Rightarrow (t_2 \ P \ st)$$

Here, st denotes a state and P denotes a predicate. We define the usual refinement relation between predicate transformers [2] by

$$t_1 \sqsubseteq t_2 \ = \ \forall P. \forall st. \ t_1 \ P \ st \Rightarrow t_2 \ P \ st$$

As a consequence of these definitions we have

$$t_1 \sqsubseteq t_2 \ = \ \forall st. \ (t_1 \sqsubseteq^I t_2) \ st$$

The example in Section 2 shows the usefulness of in-place refinement as a tool for reasoning about effect checking. In general, we expect the refinement in-place to be a useful addition to the refinement calculus.

2 Example

We have discovered the notion of in-place refinement in an effort to show the correctness of a technique for checking that a program c satisfies its effect specification s. We represent each effect as an idempotent predicate transformer s and require s to be refined by the skip command.

A simple technique to check that a program c refines an effect s is to show that c_0 refines s for every command c_0 in program c, and then use the idempotence of s. Unfortunately, this simple technique fails to show that the following program c does not modify elements of the array a for indices other than $0, 1, 2, 3, 4$:

```
c :  if (abs(j) < 3) {
         if (j > 0) {
             a[j] = 1
         } else {
             a[-2 * j] = 1
         }
     }
```

Let r denote a region of store consisting of array elements $a[0], a[1], a[2], a[3], a[4]$. It is easy to see that an execution of program c can only modify locations in r. Let $\mathsf{havoc}\ r$ denote the predicate transformer that may nondeterministically modify any location in r. For $s = \mathsf{havoc}\ r$ we then have

$$\mathsf{havoc}\ r \ \sqsubseteq\ c \tag{1}$$

However, it is not the case that $\mathsf{havoc}\ r \sqsubseteq c_0$ for every command c_0 in program c. For example, the subcommand

$$c_0 = \ (a[j] = \ 1)$$

does *not* refine $\mathsf{havoc}\ r$, i.e.

$$\mathsf{havoc}\ r \not\sqsubseteq (a[j] = \ 1)$$

The reason why the simple technique fails is that the individual commands such as c_0 are taken out of the program context, and the information about the values that variables such as j may take is lost.

To overcome limitations of the simple technique, we present a technique based on context-dependent checking of refinement. Our technique can be justified using the in-place refinement operator. To check refinement (1), we instrument program c with assert commands. If P is a predicate on program states, then command $\mathsf{assert}\ P$ denotes a command that does nothing if the initial state satisfies P, and "goes wrong" otherwise. A program that "goes wrong" terminates the execution in an undesirable way. After instrumenting program c according to our technique, we obtain program c':

```
c' : if (abs(j) < 3) {
        if (j > 0) {
            assert (havoc r ⊑^I (a[j]= 1))
            a[j]= 1
        } else {
            assert (havoc r ⊑^I (a[−2 ∗ j]= 1))
            a[−2 ∗ j]= 1
        }
     }
```

The way we make sure that the instrumented program c' does not go wrong is by proving a verification condition. Namely, in Section 5 we show that if the verification condition derived from the instrumented program is valid, then the refinement relation of form (1) holds. This is the result we intuitively expect to hold. This intuition is reflected in the fact that it is straightforward to show the result for predicate transformers generated by transition relations on states.

What we show in Section 5 is that the result holds for all conjunctive predicate transformers, including the miraculous predicate transformers [1].

By introducing the in-place refinement predicate we reduce the context-specific refinement checking to the task of checking the validity of verification conditions. Our technique can therefore easily be incorporated into a general-purpose program checking tool such as [10], which is based on verification condition validity checking and can detect a variety program errors such as null pointer dereference, array out of bounds violation, and violations of programmer-specified invariants. Modifies-clause checking is an important component of such a tool because it enables a sound, modular, and precise checking of procedure calls and method invocations.

3 Preliminaries

Let $A \rightsquigarrow B$ denote the set of partial functions from set A to set B and $A \to B$ denote total functions from A to B. We assume $(A \to B) \subseteq (A \rightsquigarrow B)$, moreover, every partial function $f \in A \rightsquigarrow B$ is a total function on its domain: $f : (\mathsf{dom} f) \to B$. We use the syntax of higher order logic, with function application denoted by juxtaposition. When writing expressions we assume that the priority of function application denoted by juxtaposition is higher than the priority of infix operators. We assume that functions are curried. We identify subsets of a set A with functions $A \to \mathsf{Bool}$. We define function override, written $\oplus$, by

$$\oplus \; : \; (A \rightsquigarrow B) \to (A \rightsquigarrow B) \to (A \rightsquigarrow B)$$

$$(g \oplus f)\; x = \begin{cases} f\; x, \text{if } f\; x \text{ is defined} \\ g\; x, \text{if } f\; x \text{ is undefined} \end{cases}$$

$$\begin{array}{rl}
\mathsf{Loc} & \textit{(set of locations)} \\
\mathsf{Value} & \textit{(set of values)} \\
\mathsf{Bool} = \{\mathsf{false}, \mathsf{true}\} & \textit{(truth values)} \\
\mathsf{State} = \mathsf{Loc} \to \mathsf{Value} & \textit{(program state)} \\
\mathsf{Update} = \mathsf{Loc} \rightsquigarrow \mathsf{Value} & \textit{(state update)} \\
\mathsf{Region} = \mathsf{Loc} \to \mathsf{Bool} & \textit{(set of locations)} \\
\mathsf{Pred} = \mathsf{State} \to \mathsf{Bool} & \textit{(predicates on states)} \\
\mathsf{Trans} = \mathsf{Pred} \to \mathsf{Pred} & \textit{(predicate transformers)}
\end{array}$$

Fig. 1. Basic Sets

Figure 1 summarizes the definitions of some basic sets of objects. We postulate a set of locations Loc and a set of values Value. We think of a location $l \in \mathsf{Loc}$ as modeling an assignable memory location of a computer store, where every location holds information representing some value $v \in \mathsf{Value}$. The precise structure of sets Loc and Value is not important for the purpose of this paper. A state $st \in \mathsf{State}$ is a total function from locations to values. A predicate $P \in \mathsf{Pred}$ is a function from states to the set of truth values Bool. We denote predicates by capital letters P, Q, possibly with subscripts. Predicates form a lattice, moreover, the lattice of predicates is a boolean algebra. We write $P_1 \subseteq P_2$ for the lattice order between predicates in the lattice. We write $P_1 \wedge P_2$, $P_1 \vee P_2$, and $\neg P_1$ for conjunction, disjunction, and negation of predicates. Each predicate transformer $t \in \mathsf{Trans}$ is a total function from predicates to predicates. Predicate transformers also form a boolean algebra.

$$
\begin{aligned}
\mathsf{assert} \ &: \ \mathsf{Pred} \rightarrow \mathsf{Trans} \\
\mathsf{assert}\ Q\ P &= Q \wedge P \\
\mathsf{assume} \ &: \ \mathsf{Pred} \rightarrow \mathsf{Trans} \\
\mathsf{assume}\ Q\ P &= \neg Q \vee P \\
\mathsf{assign} \ &: \ \mathsf{Update} \rightarrow \mathsf{Trans} \\
\mathsf{assign}\ f\ P\ st &= P(st \oplus f)
\end{aligned}
$$

Fig. 2. Basic Predicate Transformers

$$
\begin{aligned}
; \ &: \ \mathsf{Trans} \rightarrow \mathsf{Trans} \rightarrow \mathsf{Trans} \\
(t_1 \ ; \ t_2)\ P &= t_1(t_2\ P) \\
\Pi_A \ &: \ (A \rightarrow \mathsf{Trans}) \rightarrow \mathsf{Trans} \\
\Pi_A\ f\ P &= \forall a : A.\ f\ a\ P
\end{aligned}
$$

Fig. 3. Sequential Composition and Demonic Choice

Figure 2 defines basic predicate transformers $\mathsf{assert}\ Q$, $\mathsf{assume}\ Q$, and $\mathsf{assign}\ f$. We define $\mathsf{skip} = \mathsf{assume}\ \mathsf{true}$.

We build new predicate transformers from the existing ones using the following two operators:

– sequential composition ";" (function composition of predicate transformers);
– demonic choice "Π" (universal quantification).

Figure 3 shows the semantics of these two operators. Demonic choice Π_A is polymorphic in the type A of the choice set; we require $A \neq \emptyset$ and omit A from Π_A if it is clear from the context. We can define the familiar binary demonic choice $\Box$ as a special case of the unbounded choice Π, by

$$(t_1 \;\Box\; t_2) = \Pi f, \quad \text{where}$$
$$f :: \mathsf{Bool} \to \mathsf{Trans}$$
$$f\ x = \begin{cases} t_1, x = \mathsf{true} \\ t_2, x = \mathsf{false} \end{cases}$$

From the definition it follows

$$(t_1 \;\Box\; t_2)\ P = (t_1\ P) \wedge (t_2\ P)$$

Of special importance for approximating predicate transformers is the havoc command, defined as follows:

$$\mathsf{havoc} \;:\; \mathsf{Region} \to \mathsf{Trans}$$
$$\mathsf{havoc}\ r \;= \Pi(\lambda f : (r \to \mathsf{Value}).\ \mathsf{assign}\ f)$$

We say that a predicate transformer t is *positively conjunctive* iff for all functions $f : A \to \mathsf{Pred}$ where $A \neq \emptyset$

$$t(\forall x.\ f\ x) = \forall x.\ t(f\ x)$$

In this paper, the term "conjunctive" means "positively conjunctive". If a predicate transformer is conjunctive, it is also monotonic with respect to the underlying lattice order [9]. We denote the set of conjunctive predicate transformers by CTrans. All predicate transformers in Figure 2 are conjunctive. Moreover, sequential composition and demonic choice of conjunctive transformers is a conjunctive transformer.

We assume that programs are constructed from the basic predicate transformers in Figure 2 using demonic choice and sequential composition. Demonic choice with assume commands models conditional commands such as if . Conditional commands and sequential composition can express arbitrary straight-line code. Furthermore, if we assume that each procedure has a specification in terms of other predicate transformers such as havoc, we can perform conservative checking of arbitrary recursive procedures.

We call a predicate transformer s such that $s \sqsubseteq c$ an *effect of the command* c. We use the term *effect* to denote any predicate transformer s that is meant to be used as a specification for some command.

A predicate transformer s is *idempotent* iff

$$s \sqsubseteq s\,;s$$

A predicate transformer is a *may-transformer* iff

$$s \sqsubseteq \mathsf{skip}$$

Note that an idempotent may-transformer satisfies $s\,;s = s$. An idempotent effect is an effect that is an idempotent predicate transformer; a *may-effect* is an effect that is a may-transformer.

4 An Effect Checking Technique

We first show how to transform the checking of refinement $s \sqsubseteq c$ into a verification condition. The following holds:

$$\begin{aligned} & s \sqsubseteq c \\ = \; & \forall st.\ (s \sqsubseteq^I c)\ st \\ \Leftarrow \; & \forall st.\ (s \sqsubseteq^I c) st \;\wedge\; c\ \mathsf{true}\ st \\ = \; & \forall st.\ \mathsf{assert}\ (s \sqsubseteq^I c)\ (c\ \mathsf{true})\ st \\ = \; & \forall st.\ (\mathsf{assert}\ (s \sqsubseteq^I c)\,;c)\ \mathsf{true}\ st \end{aligned}$$

For a given specification s, define

$$\mathsf{check}_s\ c = \mathsf{assert}\ (s \sqsubseteq^I c)\,;c$$

We have thus reduced checking refinement $s \sqsubseteq c$ to checking whether the instrumented program

$$c' = \mathsf{check}_s\ c$$

has the property

$$c'\ \mathsf{true} = \mathsf{true} \tag{2}$$

We write simply $\mathsf{check}\ c$ instead of $\mathsf{check}_s\ c$ if the specification s is clear from the context.

We have thus obtained the following Proposition 1.

Proposition 1. *If* $(\mathsf{check}_s\ c)\ \mathsf{true} = \mathsf{true}$ *then* $s \sqsubseteq c$.

One difficulty with checking (2) is that program c may have a complicated structure, so it may be unclear how to check whether predicate $s \sqsubseteq^I c$ holds. We therefore transform the instrumented program c' into another instrumented program c'' such that:

- $c'' \sqsubseteq c'$ and
- the in-place refinement checks in c'' are of the form $s \sqsubseteq^I c_0$ where c_0 is one of the basic transformers in Figure 2.

To show (2), we attempt to prove $(c''\ \mathsf{true} = \mathsf{true})$. If $c''\ \mathsf{true} = \mathsf{true}$ holds, we conclude (2) as follows:

$$\begin{aligned} & \mathsf{true} \\ = \; & c''\ \mathsf{true} \\ \subseteq \; & c'\ \mathsf{true} \\ \subseteq \; & \mathsf{true} \end{aligned}$$

We next describe how to obtain the instrumented program c''. To obtain c'' we need to assume that the predicate transformer c is given by some syntax tree C.

Let $[\![\_]\!]$ be an interpretation function mapping syntax trees to predicate transformers. We write $\mathtt{;}$, $\mathtt{\Pi}$, $\mathtt{assert}$, $\mathtt{assume}$, $\mathtt{assign}$ for the syntax tree counterparts of ;, Π, assert, assume, assign. We thus have

$$\begin{aligned}
[\![\mathtt{;}]\!] &= ; \\
[\![\mathtt{\Pi}]\!] &= \Pi \\
[\![\mathtt{assert}]\!] &= [\![\mathsf{assert}]\!] \\
[\![\mathtt{assume}]\!] &= [\![\mathsf{assume}]\!] \\
[\![\mathtt{assign}]\!] &= [\![\mathsf{assign}]\!]
\end{aligned}$$

We extend the relation $[\![\_]\!]$ to syntax trees in the natural way:

$$\begin{aligned}
[\![C_1 \mathbin{\mathtt{;}} C_2]\!] &= [\![C_1]\!][\![\mathtt{;}]\!][\![C_2]\!] \\
[\![\mathtt{\Pi} f]\!] &= [\![\mathtt{\Pi}]\!](\lambda x.\ [\![f\ x]\!])
\end{aligned}$$

We also define the syntactic counterpart to check:

$$\mathtt{check}\ C \;=\; \mathtt{assert}\ (s \sqsubseteq^I c) \mathbin{\mathtt{;}} C$$

We next define a function instr that instruments syntax trees. Suppose that the command c is written using a syntax tree C, so that $c = [\![C]\!]$. We then let

$$c'' = [\![\mathsf{instr}\ C]\!]$$

We define the function instr by induction on the structure of a syntax tree:

$$\begin{aligned}
\mathsf{instr}\ (C_1 \mathbin{\mathtt{;}} C_2) &= (\mathsf{instr}\ C_1) \mathbin{\mathtt{;}} (\mathsf{instr}\ C_2) \\
\mathsf{instr}\ (\mathtt{\Pi} f) &= \mathtt{\Pi}(\lambda x.\ \mathsf{instr}(f\ x)) \\
\mathsf{instr}\ (\mathtt{assert}\ Q) &= \mathtt{assert}\ Q \\
\mathsf{instr}\ (\mathtt{assume}\ Q) &= \mathtt{assume}\ Q \\
\mathsf{instr}\ (\mathtt{assign}\ f) &= \mathtt{check}\ (\mathtt{assign}\ f)
\end{aligned} \tag{3}$$

To see how the instr transformation simplifies effect checking, suppose $s = \mathsf{havoc}\ r$ and $c_0 = \mathsf{assign}\ f$. Then by definition of s, c, $\mathsf{havoc}\ r$, and $\mathsf{assign}\ f$, we have:

$$\begin{aligned}
(s \sqsubseteq^I c_0)\ st \;&=\; (\mathsf{havoc}\ r \sqsubseteq^I \mathsf{assign}\ f)\ st \\
&= \forall P.\big((\forall f' \in (r \to \mathsf{Value}).\mathsf{assign}\ f'\ P\ st) \Rightarrow \\
&\qquad\qquad \mathsf{assign}\ f\ P\ st\big) \\
&\Leftarrow \mathsf{dom}\ f \subseteq r
\end{aligned}$$

To ensure that refinement $s \sqsubseteq^I c_0$ holds, it therefore suffices to check that locations assigned in the assignment command c_0 are included in the locations specified by the havoc command s.

According to our definition, instr performs all the checks at the leaves of the syntax tree. In general, we may stop the recursive application of instr and apply check at any point in the tree. To capture this idea we define a reduction relation $\mapsto$ with the property

$$\texttt{check}\ C \overset{*}{\mapsto} \mathsf{instr}\ C$$

where $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. Define first relation ρ on commands by

$$\begin{array}{rcl} \texttt{check}\ (C_1\ \texttt{;}\ C_2) & \rho & (\texttt{check}\ C_1)\ \texttt{;}(\texttt{check}\ C_2) \\ \texttt{check}\ (\Pi f) & \rho & \Pi(\lambda x.\ \texttt{check}\ (f\ x)) \\ \texttt{check}\ (\texttt{assert}\ Q) & \rho & \texttt{assert}\ Q \\ \texttt{check}\ (\texttt{assume}\ Q) & \rho & \texttt{assume}\ Q \end{array} \tag{4}$$

Next, define $\mapsto$ as the congruent closure of relation ρ i.e. define $\mapsto$ as the least relation such that

$$D[C_1] \mapsto D[C_2]$$

for all contexts $D[\,]$, and all commands C_1 and C_2 such that $C_1\, \rho\, C_2$.

5 Correctness of Effect Checking

The central result of this paper is the following theorem.

Theorem 2. *Let C_1 and C_2 be terms denoting conjunctive predicate transformers. Then*

$$C_1 \overset{*}{\mapsto} C_2$$

implies

$$[\![C_2]\!] \sqsubseteq [\![C_1]\!]$$

The rest of this section is devoted to the proof of Theorem 2.

First, since $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$, and the relation $\sqsubseteq$ is reflexive and transitive, it suffices to show that

$$C_1 \mapsto C_2$$

implies $[\![C_2]\!] \sqsubseteq [\![C_1]\!]$. We next observe that the monotonicity properties in Propositions 3, 4, 5 hold.

Proposition 3. *Let x and y be predicate transformers such that*

$$x \sqsubseteq y$$

Then for each predicate transformer z

$$x\,;z \sqsubseteq y\,;z$$

Proposition 4. *Let x and y be predicate transformers such that*

$$x \sqsubseteq y$$

Then for each monotonic predicate transformer z

$$z\,;x \sqsubseteq z\,;y$$

Proposition 5. *Let $A \neq \emptyset$ and let $f : A \to \mathsf{CTrans}$ and $g : A \to \mathsf{CTrans}$ be parameterized families of predicate transformers. If for all $a \in A$*

$$f\ a \sqsubseteq g\ a$$

then

$$\Pi f \sqsubseteq \Pi g$$

From Propositions 3, 4, 5 by induction it follows that all we need to prove is that

$$C_1\,\rho\,C_2$$

implies $[\![C_2]\!] \sqsubseteq [\![C_1]\!]$. By Definition (4), we prove the following facts:

$$(\mathsf{check}\ c_1)\,;(\mathsf{check}\ c_2) \ \sqsubseteq\ \mathsf{check}\ (c_1\,;c_2) \tag{5}$$

$$\Pi(\lambda x.\ \mathsf{check}\ (f\ x)) \ \sqsubseteq\ \mathsf{check}\ (\Pi f) \tag{6}$$

$$\mathsf{assert}\ Q \ \sqsubseteq\ \mathsf{check}\ (\mathsf{assert}\ Q) \tag{7}$$

$$\mathsf{assume}\ Q \ \sqsubseteq\ \mathsf{check}\ (\mathsf{assume}\ Q) \tag{8}$$

We proceed to show each of the properties above. Property (5) follow from Proposition 8 below. To show Proposition 8 we use Lemma 6 and Lemma 7. Lemma 6 is a simple fact used in Lemma 7.

Lemma 6. *Let s, c be predicate transformers and P a predicate. Then*

$$s\ P\ \wedge\ (s \sqsubseteq^I c)\ \subseteq\ c\ P \tag{9}$$

Proof. By applying (9) to an arbitrary state st we obtain

$$s\ P\ st \wedge (s \sqsubseteq^I c)st\ \Rightarrow c\ P\ st$$

which is a direct consequence of the definition of $\sqsubseteq^I$. ∎
Lemma 7 is the place where we need conjunctivity of commands.

Lemma 7. *Let s_2, c_1, c_2 be predicate transformers such that c_1 is conjunctive. Then*

$$c_1(s_2 \sqsubseteq^I c_2)\ \subseteq\ (c_1\,;s_2 \sqsubseteq^I c_1\,;c_2) \tag{10}$$

Proof. Because c_1 is conjunctive, c_1 is monotonic. From Lemma 6 we therefore conclude that for all predicates P

$$c_1(s_2\ P \wedge (s_2 \sqsubseteq^I c_2)) \ \subseteq\ c_1(c_2\ P) \tag{11}$$

To show (10) let st be any state satisfying

$$c_1(s_2 \sqsubseteq^I c_2)\ st \tag{12}$$

We need to show that for all predicates P,

$$c_1(s_2\ P)\ st\ \Rightarrow\ c_1(c_2\ P)\ st$$

So let P be an arbitrary predicate and assume

$$c_1(s_2\ P)\ st \tag{13}$$

Because c_1 is conjunctive, from (13) and (12) we conclude

$$c_1(s_2\ P\ \wedge\ (s_2 \sqsubseteq^I c_2))\ st$$

Now from (11) we have

$$c_1(c_2\ P)\ st$$

■

Proposition 8. *Let s_1, s_2, c_1, c_2 be predicate transformers such that c_1 is conjunctive. Then*

$$\begin{array}{c} \mathsf{assert}\ (s_1 \sqsubseteq^I c_1)\ ;\ c_1\ \ ;\ \mathsf{assert}\ (s_2 \sqsubseteq^I c_2)\ ;\ c_2 \quad \sqsubseteq \\ \mathsf{assert}\ (s_1\,;s_2 \sqsubseteq^I c_1\,;c_2)\ ;\ c_1\ \ ;\ c_2 \end{array}$$

Proof. By definition we need to show that for every predicate P and every state st

$$\begin{array}{c} (s_1 \sqsubseteq^I c_1)\ st\ \wedge\ c_1((s_2 \sqsubseteq^I c_2) \wedge (c_2\ P))\ st \quad \Rightarrow \\ (s_1\,;s_2 \sqsubseteq^I c_1\,;c_2)\ st\ \wedge\ c_1(c_2\ P)\ st \end{array} \tag{14}$$

Assume

$$(s_1 \sqsubseteq^I c_1)\ st \tag{15}$$

and

$$c_1((s_2 \sqsubseteq^I c_2) \wedge (c_2\ P))\ st \tag{16}$$

Because c_1 is conjunctive, c_1 is monotonic, so from (16) we conclude

$$c_1(c_2\ P)\ st$$

which is the second conjunct in the conclusion of (14). It remains to show the first conjunct i.e. that for every predicate P_1,

$$s_1(s_2\ P_1)\ st\ \Rightarrow c_1(c_2\ P_1)\ st \tag{17}$$

Let P_1 be an arbitrary predicate. From (16) and monotonicity of c_1 we conclude

$$c_1(s_2 \sqsubseteq^I c_2)\ st$$

Applying Lemma 7 we conclude

$$(c_1\,; s_2 \sqsubseteq^I c_1\,; c_2)\ st$$

which implies

$$c_1(s_2\ P_1)\ st \Rightarrow c_1(c_2\ P_1)\ st \tag{18}$$

On the other hand, from the assumption (15) we conclude

$$s_1(s_2\ P_1)\ st \Rightarrow c_1(s_2\ P_1)\ st \tag{19}$$

From (18) and (19) we conclude (17). ∎

The following Corollary 9 follows from Proposition 8 by taking $s_1 = s_2 = s$ where s is idempotent.

Corollary 9. *Let s, c_1, c_2 be predicate transformers such that c_1 is conjunctive and*

$$s\,; s \sqsubseteq s$$

Then

$$\begin{gathered}\mathsf{assert}\ (s \sqsubseteq^I c_1)\ ;\ c_1\ ;\ \mathsf{assert}\ (s \sqsubseteq^I c_2)\ ;\ c_2 \quad \sqsubseteq \\ \mathsf{assert}\ (s \sqsubseteq^I c_1\,; c_2)\ ;\ c_1\ ;\ c_2\end{gathered}$$

This completes the proof of Property (5).

Property (6) follows from the following proposition.

Proposition 10. *Let s be a predicate transformer and $f : A \to \mathsf{Trans}$ an indexed family of predicate transformers. Then*

$$\begin{aligned}&\Pi(\lambda x.\ \mathsf{assert}\ (s \sqsubseteq^I f\ x)\ ;(f\ x))\\ =\ &\mathsf{assert}\ (s \sqsubseteq^I \Pi f)\ ;\ \Pi f\end{aligned}$$

Proof. Let P be an arbitrary predicate and st an arbitrary state. By definition it suffices to show

$$\begin{aligned}&\forall x.\ (s \sqsubseteq^I f\ x)\ st\ \wedge\ f\ x\ P\ st\\ =\ &(s \sqsubseteq^I \Pi f)\ st\ \wedge\ (\Pi f)\ P\ st\end{aligned}$$

We have

$$\begin{aligned}&\forall x.\ (s \sqsubseteq^I f\ x)\ st\ \wedge\ f\ x\ P\ st\\ =\ &\forall x.\ (\forall P'.\ s\ P'\ st\ \Rightarrow f\ x\ P'\ st)\ \wedge\ f\ x\ P\ st\\ =\ &(\forall P'.\ s\ P'\ st\ \Rightarrow \forall x.\ f\ x\ P'\ st)\ \wedge\ \forall x.\ f\ x\ P\ st\\ =\ &(\forall P'.\ s\ P'\ st\ \Rightarrow (\Pi f)\ P'\ st)\ \wedge\ (\Pi f)\ P\ st\\ =\ &(s \sqsubseteq^I \Pi f)\ st\ \wedge\ (\Pi f)\ P\ st\end{aligned}$$

∎

To show Theorem 2 it remains to show that we may simply drop the instrumentation check in front of assert and assume commands. Here we use the assumption that s is a may-effect.

Proposition 11. *Let s be a predicate transformer such that*

$$s \sqsubseteq \mathsf{skip}$$

and let Q be a predicate. Then

$$\mathsf{assume}\ Q = \mathsf{assert}\ (s \sqsubseteq^I (\mathsf{assume}\ Q))\ ;\ (\mathsf{assume}\ Q)$$

Proof. Because

$$s \sqsubseteq \mathsf{skip} \sqsubseteq \mathsf{assume}\ Q$$

we have

$$(s \sqsubseteq^I \mathsf{assume}\ Q) = \mathsf{true}$$

so

$$\mathsf{assert}\ (s \sqsubseteq^I \mathsf{assume}\ Q) = \mathsf{skip}$$

■

Proposition 12. *Let s be a predicate transformer such that*

$$s \sqsubseteq \mathsf{skip}$$

and let Q be a predicate. Then

$$\mathsf{assert}\ Q =$$

$$\mathsf{assert}\ (s \sqsubseteq^I \mathsf{assert}\ Q)\ ;\ \mathsf{assert}\ Q$$

Proof. Let P be an arbitrary predicate. We need to show

$$Q \wedge P = (s \sqsubseteq^I \mathsf{assert}\ Q) \wedge Q \wedge P$$

which is equivalent to

$$(Q \wedge P)\ \subseteq\ (s \sqsubseteq^I \mathsf{assert}\ Q)$$

Let st be an arbitrary state. Assume $(Q\ st)$ and $(P\ st)$. We show that for all predicates P'

$$s\ P'\ st\ \Rightarrow\ Q\ st\ \wedge\ P'\ st \tag{20}$$

Let P' be a predicate such that $(s\ P'\ st)$. Because $(s \sqsubseteq \mathsf{skip})$, we conclude $(P'\ st)$. We have previously assumed $(Q\ st)$, so

$$Q\ st\ \wedge\ P'\ st$$

Hence, (20) holds. ■

We have thus completed the proof of Theorem 2.

We can summarize the correctness of our technique as follows. Let C be a syntax tree, let $c = [\![C]\!]$, and let s be an idempotent may effect. Let $C_1 = \mathtt{check}\ C$, let $C_1 \overset{*}{\mapsto} C_2$, and let $c'' = [\![C_2]\!]$. If $c''\ \mathsf{true} = \mathsf{true}$, then $[\![\mathtt{check}\ C]\!]\ \mathsf{true} = (\mathsf{check}\ c)\ \mathsf{true} = \mathsf{true}$ by Theorem 2, so $s \sqsubseteq c$ by Proposition 1.

6 Some Consequences

An important example of an idempotent may-effect s is $s = \mathsf{havoc}\ r$.

Proposition 13. *Let r be an arbitrary set of locations. Then*

$$\mathsf{havoc}\ r \ \sqsubseteq\ \mathsf{skip}$$

and

$$\mathsf{havoc}\ r \ \sqsubseteq\ \mathsf{havoc}\ r\ \ ;\ \mathsf{havoc}\ r$$

We next exhibit a slightly more general form of an idempotent may-effect. First we show Lemma 14 that allows canceling of assume and assert statements.

Lemma 14. *Let Q_0 and Q_1 be predicates. Then*

$$\mathsf{skip}\ \sqsubseteq\ \mathsf{assume}\ Q_1\ \ ;\ \mathsf{assert}\ Q_0 \tag{21}$$

iff

$$Q_1 \subseteq Q_0$$

Proof. By definition, (21) holds iff

$$\forall P.\ P \subseteq (\neg Q_1 \vee (Q_0 \wedge P)) \tag{22}$$

which can be easily shown equivalent to $Q_1 \subseteq Q_0$. ∎

Proposition 15. *Let*

$$s = \quad \mathsf{assert}\ Q_0\ \ ;\ \mathsf{havoc}\ r\ \ ;\ \mathsf{assume}\ Q_1$$

where r is a set of locations and Q_0 and Q_1 are predicates such that $Q_1 \subseteq Q_0$. Then

$$s\ \sqsubseteq\ s\,;s$$

Proof. By Lemma 14 and Proposition 13. ∎

Proposition 16. *Let*

$$s = \quad \mathsf{assert}\ Q_0\ \ ;\ \mathsf{havoc}\ r\ \ ;\ \mathsf{assume}\ Q_1$$

where r is a set of locations and Q_0 and Q_1 are predicates such that $Q_0 \subseteq Q_1$. Then

$$s \sqsubseteq \mathsf{skip}$$

Proof. By shunting rules [2, Page 223], $s \sqsubseteq \mathsf{skip}$ is equivalent to

$$\mathsf{havoc}\ r\ \ \sqsubseteq\ \ \mathsf{assume}\ Q_0\ \ ;\ \mathsf{skip}\ \ ;\ \mathsf{assert}\ Q_1$$

The result then follows by Lemma 14 and Proposition 13. ∎

From Proposition 15 and Proposition 16 we obtain the following Corollary 17.

Corollary 17. *Let Q be a predicate and r a set of locations. Then*

$$s = \quad \mathsf{assert}\ Q\ ;\ \mathsf{havoc}\ r\ ;\ \mathsf{assume}\ Q$$

is an idempotent may-effect.

Corollary 17 and Theorem 2 imply that our technique for effect checking is applicable to the effects of the form

$$s = \quad \mathsf{assert}\ Q\ ;\ \mathsf{havoc}\ r\ ;\ \mathsf{assume}\ Q$$

Such effects capture the following idea: if all commands preserve the invariant Q and change only locations in r, then the entire program preserves the invariant Q and changes only locations in r.

We next give some simple rules for constructing effects.

Proposition 18. *Let s_1 and s_2 be idempotent effects such that:*

$$s_1 ; s_2 \ \sqsubseteq \ s_2 ; s_1$$

Then

$$s = s_1 ; s_2$$

is an idempotent effect as well. Moreover, if $s_1 \sqsubseteq \mathsf{skip}$ and $s_2 \sqsubseteq \mathsf{skip}$ then $s \sqsubseteq \mathsf{skip}$ as well.

Define Kleene iteration s^* of a predicate transformer s as a demonic choice of all finite sequential compositions of s. More precisely, given an effect s, define $f : \mathsf{Nat} \to \mathsf{Trans}$ where Nat is the set of nonnegative integers by

$$f\ 0 = \mathsf{skip}$$
$$f\ (k+1) = s ;(f\ k)$$

and let

$$s^* = \Pi f$$

Proposition 19. *Let s be positively conjunctive transformer. Then s^* is an idempotent may-effect.*

Proof. Clearly, s^* is a may-effect because it is a demonic choice with skip. For idempotence, show

$$s^* \sqsubseteq s ; s^*$$

and then use induction and lattice properties. ■

7 Related Work

[1] and [18] contain a systematic introduction to the refinement calculus. [5,19] present applications of the refinement calculus to program derivation. To the best of our knowledge, we are the first to introduce the notion of in-place refinement into the refinement calculus.

In this paper we have shown that specification commands [17] of a special form can be used as effects and checked against a program on a per-command basis.

[12] uses a type system containing effects as elements of a commutative idempotent algebra, which is similar to our requirement on idempotent may-effects. Type systems supporting effect checking in object-oriented programs include [3,4,6,11].

In contrast to most type system approaches for effect checking, our effect checking approach is flow-sensitive. Flow-sensitivity is essential for dealing with aliasing in an object-oriented programming language. [15] uses an instance of the technique described in this paper to check modifies clauses. Another flow-sensitive approach is role analysis [13], which uses effects to enable compositional analysis of properties of objects that move between data structures.

An alternative to the technique in this paper is to use a specialized program analysis and express the analysis result as an instrumentation of the program with assumption $\mathsf{assume}\ Q$. Each instrumentation commands $\mathsf{assume}\ Q$ describes an approximation of the set of reachable states at a program point. A program analysis can be cast into this framework using *refinement in context*, [1, Page 463]. Suppose that s is an idempotent may-effect. Even if it is not the case that $s \sqsubseteq c_0$ for every command c_0 of the program, if the bound on reachable states Q is precise enough, it may be possible to show $s \sqsubseteq (\mathsf{assume}\ Q); c_0$. If this relationship holds for all program points, and s is an idempotent may-effect, then monotonicity of nondeterministic choice and sequential composition allow us to conclude that the entire program refines the effect s. This reasoning can be used to explain correctness of program analyses such as [13] that use specialized techniques to check procedure effects.

8 Conclusion

Checking refinement is a difficult problem in general. However, if the specification is of a special form, we can automate such checks. In this paper we presented a technique for showing refinement in the case when the specification is an idempotent may-effect. We have implemented a modifies clause checker based on this technique, making use of a theorem prover tailored for program checking [7,10]. Our initial experience with the modifies clause checker suggests that the annotation burden of specifying side effects is acceptable.

We have found the refinement calculus to be a useful framework for reasoning about program checking. To show correctness of our approach to effect checking we introduced a new refinement calculus construct, in-place refinement. In-place refinement allows refinement checking to be incorporated into the checking of verification conditions, resulting in a technique that takes into account program control-flow as well as the conditions of conditional commands. We have shown the correctness of our technique when programs are conjunctive predicate transformer and specifications are idempotent may-effects. This general characterization permits various representations for the effect checking, allowing the abstraction of program store locations to be tailored for the desired application.

Acknowledgements We would like to thank Ralph Back for useful discussions. We thank Chandrasekhar Boyapati and Patrick Lam for useful comments on a draft of this paper.

References

1. Ralph-Johan Back and Joakim von Wright. Combining angels, demons and miracles in program specifications. *Theoretical Computer Science*, 100(2):365–383, 1992.
2. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, 1998.
3. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. A type system for preventing data races and deadlocks. In *Proc. 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.
4. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proc. 30th ACM POPL*, 2003.
5. Michael Butler, Jim Grundy, Thomas Langbacka, Rimvydas Ruksenas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In *Proc. Formal Methods Pacific '97*, 1997.
6. Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.
7. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
8. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.
9. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.
11. Aaron Greenhouse and John Boyland. An object-oriented effects system. In *Proc. 13th European Conference on Object-Oriented Programming*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.
12. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.
13. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th ACM POPL*, 2002.
14. K. Rustan M. Leino and Rajit Manohar. Joining specification statements. *Theoretical Computer Science*, 216:375–394, 1999.

15. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.
16. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999–002, COMPAQ Systems Research Center, 1999.
17. Carroll Morgan. The specification statement. *Transactions on Programming Languages and Systems*, 10(3), 1988.
18. Carroll Morgan. *Programming from Specifications (2nd ed.)*. Prentice-Hall, Inc., 1994.
19. Joakim von Wright. Program refinement by theorem prover. In *Proc. 6th BCS-FACS Refinement Workshop*, 1994.