Second-Order Programs with Preconditions

Markus Aderhold

Technische Universität Darmstadt, Germany
aderhold@informatik.tu-darmstadt.de

Abstract. In the implementation of procedures, developers often assume that the input satisfies certain properties; for example, *binary search* assumes the array to be sorted. Such requirements on the input can be formally expressed as preconditions of procedures. If a second-order procedure p (e.g., *map* or *foldl*) is called with a first-order procedure f that has a precondition, the question arises whether p will call f only with arguments that satisfy the precondition of f. In this paper, we propose a method to statically analyze if all procedure calls in a given second-order program satisfy the respective preconditions. In particular, we consider indirect calls of procedures that are passed as an argument to a second-order procedure.

1 Introduction

One key feature of many algorithms is that they solve a given problem for a wide range of input values. For instance, the *quicksort* algorithm is able to sort lists of arbitrary length. Often, however, algorithms are *not* intended to work for completely arbitrary inputs, but expect the input to satisfy certain preconditions. For example, *binary search* requires a *sorted* array, and *division* requires the divisor to be different from zero.

When a program applies an (implementation of an) algorithm to arguments that violate the preconditions of the algorithm, this usually leads to undesirable behavior: The program raises an exception (e.g., division by zero) or produces an unexpected result (e.g., binary search in general fails to find an element in an unsorted array).

As a first step to finding program errors that are due to violated preconditions, it is helpful to write down the preconditions explicitly and formally. Some programming languages such as Eiffel [9] or Spec# [4] offer dedicated constructs to specify so-called *contracts*. For instance, a contract can guarantee that a procedure provides a certain functionality (expressed by a postcondition) provided that the caller supplies appropriate input (expressed by a precondition).

If the preconditions of all procedures in a program are formally specified, one can then try to apply techniques for the static analysis of programs to verify that the preconditions of all procedures are satisfied during any conceivable run of the program. The basic idea is to ensure that for each procedure call $p(t_1, \ldots, t_n)$ in the program, the arguments $t_1, \ldots, t_n$ satisfy the precondition of p. For example,

S. Siegler and N. Wasser (Eds.): Walther Festschrift, LNAI 6463, pp. 129–143, 2010.

for each occurrence of a procedure call $div(t_1, t_2)$, one needs to show that the divisor t_2 is different from zero.

In this paper, we consider *second-order* programs with preconditions. A typical example of a second-order procedure is *map*, which takes a function f as well as a list l and returns the list that results from l by applying f to each element of l. Since f may have a precondition, say c_f, a procedure call $map(f, l)$ is only safe (in the sense that executing it will not violate any preconditions) if $c_f[x]$ is satisfied for all elements x of list l:

$$\forall x \in l.\ c_f[x] \tag{1}$$

The technique we propose in this paper automatically generates a formula equivalent to (1) just from the "raw definition" of *map*. In other words, for second-order procedures developers need not specify preconditions that propagate the preconditions of their function parameters (such as precondition (1) for *map*); instead, our technique automatically generates verification conditions to statically analyze if the preconditions of function parameters are respected.

Our technique has been implemented and integrated into √`eriFun`, a semi-automated verifier for functional programs [17,18]. √`eriFun`'s input language $\mathcal{L}$ [16] for functional programs allows developers to annotate procedures with preconditions [14]. √`eriFun` generates verification conditions to verify that the precondition of each procedure call in a program is satisfied. Semantically, a procedure with a precondition is considered as an incompletely defined procedure [19] such that the return value is indetermined if the precondition is violated.

The two main contributions of this paper are the following:

– We simplify the semantics of incompletely defined first-order procedures that Walther and Schweitzer proposed in [19]. By treating such procedures as procedures with implicit preconditions, we combine the advantages of [19] and [14] and get a unified view on both concepts.
– We describe a method to statically analyze if preconditions are satisfied in *second-order programs*. For second-order procedures, this method automatically identifies which arguments the function parameter will be applied to and generates the corresponding verification conditions to show that the precondition is satisfied for these calls.

In Sect. 2, we introduce the programming language $\mathcal{L}$ that is used in √`eriFun` and we present the simplified semantics of first-order polymorphic procedures with preconditions. Sect. 3 extends syntax and semantics to second-order programs. In Sect. 4, we describe our method to statically analyze if the preconditions of all functions that may be called during the evaluation of a given term are satisfied. We point to related work in Sect. 5 and conclude with a discussion of our results in Sect. 6.

2 First-Order Programs

The input language $\mathcal{L}$ of √`eriFun` that we consider in this section roughly corresponds to the first-order fragment of ML or Haskell with strict evaluation [14,16]. First, we describe the syntax of first-order programs with preconditions (Sect. 2.1). Then we define the semantics of such programs (Sect. 2.2). We show how incompletely defined programs [19] fit into this approach (Sect. 2.3) and compare it with the original definitions described in [14,19] (Sect. 2.4).

In our description of $\mathcal{L}$ we omit some technical details and some syntactical constructions that are not needed in the context of this paper. The interested reader is invited to consult [1,16] for additional information.

2.1 Syntax

$\mathcal{L}$ offers definition principles for freely generated polymorphic data types, for first-order procedures that operate on these data types, and for statements about the data types and procedures. A *base type* is a type variable $@A$ or an expression of the form $str[\tau_1, \ldots, \tau_k]$, where $\tau_1, \ldots, \tau_k$ are base types and *str* is a k-ary type constructor ($k \geq 0$). *Type constructors* are defined by expressions of the following form:

$$\texttt{structure}\ str[@A_1, \ldots, @A_k] <= \ \ldots,\ cons(sel_1 : \tau_1, \ldots, sel_n : \tau_n),\ \ldots \quad (2)$$

The τ_j are base types, and *str* may only occur as $str[@A_1, \ldots, @A_k]$ in the τ_j. Each *cons* is called a *data constructor* and the sel_j are called *selectors.* We write $\mathcal{S}_{cons} := \{sel_1, \ldots, sel_n\}$ for the set of selectors that belong to data constructor *cons*. At least one data constructor of a type constructor definition needs to be *irreflexive*, which means that *str* does not occur in $\tau_1, \ldots, \tau_n$. An expression $?cons(t)$ checks if t denotes a value of the form $cons(\ldots)$.

Let $\Sigma(P)$ denote the signature of all function symbols defined by an $\mathcal{L}$-program P, including function symbols = for equality and *if* for case analyses. As usual, $\mathcal{T}(\Sigma(P), \mathcal{V})$ denotes the set of all *terms* over $\Sigma(P)$ and a set $\mathcal{V}$ of variables. We write $\mathcal{T}(\Sigma(P))$ instead of $\mathcal{T}(\Sigma(P), \emptyset)$ for the set of all *ground terms* over $\Sigma(P)$. $\Sigma(P)^c \subset \Sigma(P)$ contains all data constructors of P. A *literal* is an *if*-free Boolean term or the negation *if b then false else true* of such a term; e.g., $x \neq y$ abbreviates *if* $x = y$ *then false else true.*

A procedure is defined by an expression of the form

$$\texttt{procedure}\ proc(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau <= \texttt{assume}\ c_{proc};\ B_{proc} \quad (3)$$

where $c_{proc} \in \mathcal{T}(\Sigma(P), \{x_1, \ldots, x_n\})$ is a Boolean term that specifies the precondition of *proc*, also called the *context requirement* of *proc*. Omitting the context requirement is equivalent to specifying $c_{proc} := true$. The *body* of *proc* is a term $B_{proc} \in \mathcal{T}(\Sigma(P) \cup \{proc\}, \{x_1, \ldots, x_n\})$.

The example program in Fig. 1 defines data types *bool*, $\mathbb{N}$, and $list[@A]$ by enumerating the respective data constructors *true*, *false*, 0, *succ*, ø, and :: as well as the corresponding selectors; e.g., selector *pred* denotes the predecessor function. The context requirements of procedures "/" (for division) and "!!" (for list access by index) express that the divisor must not be zero and that n must be an index within list l, respectively.

```
structure bool <= true, false
structure ℕ <= 0, succ(pred : ℕ)
structure list[@A] <= ø, ::(hd : @A, tl : list[@A])
procedure [infix] >(x, y : ℕ) : bool <=
if x = 0 then false else if y = 0 then true else pred(x) > pred(y)
procedure [infix] −(x, y : ℕ) : ℕ <=
if x = 0 then 0 else if y = 0 then x else pred(x) − pred(y)
procedure [infix] /(x, y : ℕ) : ℕ <=
assume y ≠ 0;  if y > x then 0 else succ((x − y)/y)
procedure len(l : list[@A]) : ℕ <=
if l = ø then 0 else succ(len(tl(l)))
procedure [infix] !!(l : list[@A], n : ℕ) : @A <=
assume len(l) > n;  if n = 0 then hd(l) else tl(l) !! pred(n)
```

Fig. 1. A first-order program with preconditions in procedures "/" and "!!"

2.2 Semantics

For a program P and a ground type τ, $\mathbb{V}(P)_\tau$ denotes the *values of type* τ: For each ground base type τ (i.e., $\tau = str[\tau_1, \dots, \tau_k]$ such that no type variables occur in $\tau_1, \dots, \tau_k$), $\mathbb{V}(P)_\tau := \mathcal{T}(\Sigma(P)^c)_\tau$. For each ground base type τ there exists at least one value $\omega_\tau \in \mathbb{V}(P)_\tau$. These so-called *witness values* can be obtained by the following definition that simply extends the construction from [15] to polymorphic type constructors:

Definition 1. *For a ground base type τ, the* witness value $\omega_\tau \in \mathbb{V}(P)_\tau$ *is defined by $\omega_\tau := false$ if $\tau = bool$ and by $\omega_\tau := cons(\omega_{\theta(\tau_1)}, \dots, \omega_{\theta(\tau_n)})$ if $\tau = str[\tau'_1, \dots, \tau'_k]$ for a type constructor as in (2), where $\theta := \{@A_1/\tau'_1, \dots, @A_k/\tau'_k\}$ and "cons" is the first data constructor of "str" such that $\mathcal{S}_{cons} = \emptyset$ or—if such a data constructor does not exist—such that "cons" is irreflexive.*

The call-by-value interpreter $\mathsf{eval}_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)$ defines the operational semantics of $\mathcal{L}$ by mapping ground terms $t \in \mathcal{T}(\Sigma(P))_\tau$ to values $\mathsf{eval}_P(t) \in \mathbb{V}(P)_\tau$. It is a partial function, because some procedures in program P may not terminate. The computation steps of the interpreter eval_P are defined by the so-called *computation calculus*:

Definition 2. *The language of the computation calculus is $\mathcal{T}(\Sigma(P))$. The inference rules of the computation calculus (called* computation rules*) are of the form "$\frac{t}{t'}$, if $\mathsf{cond}[t]$" for a side condition $\mathsf{cond}[t]$ such that $t' \in \mathcal{T}(\Sigma(P))_\tau$ whenever $t \in \mathcal{T}(\Sigma(P))_\tau$ for some ground type τ. We write $t \Rightarrow_P t'$ iff t' results from t by applying some computation rule. $\Rightarrow_P^*$ denotes the reflexive and transitive closure of $\Rightarrow_P$. We write $t \Rightarrow_P^! t'$ iff $t \Rightarrow_P^* t'$ and $t' \not\Rightarrow_P t''$ for all $t'' \in \mathcal{T}(\Sigma(P))$.*

Fig. 2 presents a selection of computation rules (see [1] for an exhaustive list), where *cons* and *cons'* are data constructors and $q_i \in \mathbb{V}(P)$ for all i. To

$$(1)\ \frac{?cons(cons(q_1,\ldots,q_n))}{true} \qquad (2)\ \frac{?cons'(cons(q_1,\ldots,q_n))}{false}\ , \text{ if } cons' \neq cons$$

$$(3)\ \frac{sel_i(cons(q_1,\ldots,q_n))}{q_i}\ , \text{ if selector } sel_i \in \mathcal{S}_{cons} \qquad (4)\ \frac{sel'(cons(q_1,\ldots,q_n))}{\omega_\tau}\ , \text{ if selector } sel' \notin \mathcal{S}_{cons}$$

$$(5)\ \frac{q_1 \mathtt{=} q_2}{true}\ , \text{if } q_1 = q_2 \qquad (6)\ \frac{q_1 \mathtt{=} q_2}{false}\ , \text{if } q_1 \neq q_2$$

$$(7)\ \frac{proc(q_1,\ldots,q_n)}{\big(if\ c_{proc}\ then\ B_{proc}\ else\ \omega_\tau\big)[x_1/q_1,\ldots,x_n/q_n]}$$

Fig. 2. Selected inference rules of the computation calculus

each $t \in \mathcal{T}(\Sigma(P))$ at most one computation rule is applicable and $t \Rightarrow^!_P t'$ implies $t' \in \mathbb{V}(P)$ [1]. Thus we define the interpreter by

$$\mathsf{eval}_P(t) := \begin{cases} t' & \text{if } t \Rightarrow^!_P t' \text{ for some } t' \in \mathbb{V}(P) \\ \bot & \text{if } t \not\Rightarrow^!_P t' \text{ for all } t' \in \mathbb{V}(P)\,. \end{cases}$$

A universally quantified formula of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$, where $b \in \mathcal{T}(\Sigma(P), \mathcal{V})_{bool}$, is *true* iff all procedures in P terminate and $\mathsf{eval}_{P'}(b[q_1,\ldots,q_n]) = true$ for each terminating program $P' \supseteq P$ and all $q_1,\ldots,q_n \in \mathbb{V}(P')$. Program P' may define additional data types and procedures, so this definition of truth ensures monotonicity in the sense that a formula remains true when program P is extended.

Computation rules (1)–(3) and (5)–(6) are straightforward and coincide with the semantics considered in [19]. Computation rule (4) differs from [19], however: If a selector *sel′* that does not belong to data constructor *cons* is applied to a value $cons(\ldots)$, e.g., $pred(\mathtt{0})$ or $hd(\varnothing)$, Walther and Schweitzer propose to consider the result as an *indetermined* value and do not evaluate such terms further. A so-called *fair completion* [19] resolves the indeterminism by defining a witness term for such “inappropriate” applications of selectors.[1] When reasoning about programs, all such fair completions need to be considered, so neither $pred(\mathtt{0}) \mathtt{=} \mathtt{0}$ nor $pred(\mathtt{0}) \mathtt{\neq} \mathtt{0}$ are true formulas, because $pred(\mathtt{0})$ is indetermined. Since subsequent extensions [14] of √eriFun demand that $?cons(t)$ be verified for each selector call $sel(t)$, where *cons* is the constructor that *sel* belongs to, such indetermined values cannot occur anymore. Therefore we simplify the semantics as in computation rule (4), which returns the fixed witness value ω_τ.

Computation rule (7) is also simplified in this spirit. We will discuss this rule further in Sect. 2.4. The idea is that a procedure call is evaluated to the

[1] In addition, a fair completion specifies the result of a procedure if its context requirement is violated. “Fairness” means that the completion does not affect the termination behavior of procedures.

```
procedure [infix] $!!(l : list[@A],\ n : \mathbb{N}) : @A$ <=
  if $l = \emptyset$ then $\star$ else if $n = 0$ then $hd(l)$ else $tl(l)\,!!\,pred(n)$
procedure $\nabla_{!!}(l : list[@A],\ n : \mathbb{N}) : bool$ <=
  if $l = \emptyset$ then false else if $n = 0$ then true else $\nabla_{!!}(tl(l), pred(n))$
```

Fig. 3. Alternative implementation of procedure "!!"

witness value of the respective type if the context requirement of the procedure is violated.

2.3 Incompletely Defined Procedures

For some procedures, one needs to implement auxiliary procedures to formulate a suitable context requirement. For instance, procedure "!!" in Fig. 1 uses procedures ">" and "*len*" to express the precondition on its parameters. Instead of formulating the context requirement (including auxiliary procedures) explicitly, one can define procedure "!!" *incompletely* by leaving some cases *underspecified* [6].

Fig. 3 shows an alternative implementation of procedure "!!". Symbol "$\star$" denotes an undefined return value; in other words, this case "should not occur". Walther and Schweitzer [19] show how a *domain procedure* ∇_f can be synthesized for each incompletely defined procedure f. This domain procedure is completely defined and returns *true* if and only if f is determined for the given input. For example, $\nabla_{!!}(l, n)$ returns *true* if and only if n is smaller than the length of list l. Hence one can use $\nabla_{!!}(l, n)$ as an implicit context requirement for procedure "!!" without having to implement procedures ">" and "*len*".

2.4 Discussion

In the previous subsections we described two concepts to implement procedures with preconditions. The first possibility is to specify a context requirement for a procedure explicitly [14]. The second possibility is to define a procedure incompletely by marking certain cases as undefined [19]; this implicitly induces a precondition, namely that the input must not lead to the undefined results.

In √eriFun, support for incompletely defined procedures was added first [19]. The semantics of procedures with context requirements was then defined by reducing them to incompletely defined procedures [14]. Incompletely defined procedures were *not* considered as procedures with (implicit) preconditions, so they could be called without inducing any verification conditions. Only if a precondition was explicitly specified did the verifier generate the verification condition that the context requirement be satisfied for each call of the procedure.

In the previous subsections, we presented a different view on these two concepts. We introduced procedures with context requirements as the basic concept and considered incompletely defined procedure as syntactical sugar that may

```
procedure $map(f : @A \to @B,\ l : list[@A]) : list[@B]$ <=
$if\ l = \emptyset\ then\ \emptyset\ else\ f(hd(l)) :: map(f, tl(l))$
procedure $foldl(f : @A \times @B \to @A,\ x : @A,\ l : list[@B]) : @A$ <=
$if\ l = \emptyset\ then\ x\ else\ foldl(f,\ f(x, hd(l)),\ tl(l))$
procedure $every(f : @A \to bool,\ l : list[@A]) : bool$ <=
$if\ l = \emptyset\ then\ true\ else\ if\ f(hd(l))\ then\ every(f, tl(l))\ else\ false$
```

Fig. 4. Second-order procedures

simplify the definition of procedures by the automated synthesis of domain procedures as implicit preconditions.

Regarding the theoretical treatment, our view has the benefit that several definitions become considerably simpler. For instance, in [19] the computation rule to execute procedure calls is about three times as long as our computation rule (7), because it needs to take indetermined values into account. Since we will statically enforce that all preconditions (both explicit ones and implicit ones via incomplete definitions) are satisfied in a program, we can simply designate the concrete value ω_τ as the result of a procedure call that violates the precondition c_{proc}. In practice, the advantage of the unified view is that indetermined values may no longer occur in program executions, which eliminates a frequent cause of errors.

3 Second-Order Programs

We define the order $o(\tau)$ of base types τ such as $\mathbb{N}$ or $list[\mathbb{N}]$ as 0; the order of a function type $\tau_1 \times \ldots \times \tau_n \to \tau$ is $1 + \max_i o(\tau_i)$ for a base type τ [3]. A *type* is a base type or an expression of the form $\tau_1 \times \ldots \times \tau_k \to \tau$ for types $\tau_1, \ldots, \tau_k, \tau$.

The procedures in Fig. 1 are *first-order* procedures, because their type has order 1. A *second-order* procedure takes one or more first-order functions as parameters. Fig. 4 shows some common examples of second-order procedures.

The semantics of second-order programs is defined by extending the definitions from Sect. 2.2 in the following way: For each ground type of the form $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, the set $\mathbb{V}(P)_\tau$ of values of type τ contains all closed λ-expressions of type τ; e.g., $\lambda l : list[\mathbb{N}].\ len(l) \in \mathbb{V}(P)_{list[\mathbb{N}]\to\mathbb{N}}$. The witness value of such a type τ is defined as $\omega_\tau := \lambda x_1 : \tau_1, \ldots, x_k : \tau_k.\ \omega_{\tau_{k+1}}$. The computation calculus is extended by computation rule

$$(8)\quad \frac{(\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\ t)(q_1, \ldots, q_n)}{t[x_1/q_1, \ldots, x_n/q_n]}\ \text{, if } q_1, \ldots, q_n \in \mathbb{V}(P)$$

to facilitate β-reduction of terms.

We implicitly assume procedure bodies to be in η-long form; e.g., $map(f, tl(l))$ abbreviates $map(\lambda z : @A.\ f(z),\ tl(l))$ in Fig. 4, because $f =_\eta \lambda z : @A.\ f(z)$.

4 Static Analysis of Second-Order Programs

Given a second-order program, we would like to find out if the preconditions of all procedures are satisfied during any conceivable run of the program. For a procedure call t_1/t_2 we obviously need to show $t_2 \neq \mathsf{0}$ according to the context requirement of procedure "/". But what is a suitable verification condition for an indirect call of "/" as in $map(\lambda n : \mathbb{N}.\, c/n,\, l)$, for example? Since *map* will call c/n only for values $n \in l$, it would be sufficient to show "$n \neq \mathsf{0}$ for all $n \in l$". In this verification condition, "$n \neq \mathsf{0}$" comes from the context requirement of procedure "/", while the quantification "for all $n \in l$" comes from our knowledge about the behavior of *map*.

In Sect. 4.1, we describe the concept of *quantification procedures* that were introduced in [2] to synthesize induction axioms for procedures that involve second-order procedures. In Sect. 4.2, we present our method to uniformly generate verification conditions for terms that occur in second-order programs; here we use quantification procedures when procedures are passed as arguments to a second-order procedure in order to capture the behavior of the second-order procedure. We illustrate the benefits of our approach in Sect. 4.3 and give a medium-sized example in Sect. 4.4.

4.1 Quantification Procedures

For each second-order procedure *proc* with a function parameter f, one can uniformly synthesize a quantification procedure *forall.proc* that checks whether some predicate p holds for all arguments that are passed to f by *proc*. The corresponding definition in [2] considered second-order procedures *without* preconditions, so we generalize this definition as follows:

Definition 3. *For each second-order procedure*

```
procedure $proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc}$ <=
assume $c_{proc}$; $B_{proc}$
```

the quantification procedure *forall.proc* for *proc is defined by*

```
procedure $forall.proc(p : \tau_1 \times \ldots \times \tau_m \to bool,$
                       $f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : bool$ <=
$\mathsf{ALL}_f(c_{proc}) \wedge \textit{if}\ c_{proc}\ \textit{then}\ \mathsf{ALL}_f(B_{proc})\ \textit{else true}$
```

where

$$\mathsf{ALL}_f(v) := \textit{true}$$
$$\mathsf{ALL}_f(f(t_1, \ldots, t_m)) := p(t_1, \ldots, t_m) \wedge \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_m)$$
$$\mathsf{ALL}_f(g(t_1, \ldots, t_n)) := \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_n)$$
$$\mathsf{ALL}_f(h(\lambda \boldsymbol{y}.\, t_0,\, t_1)) := \mathsf{ALL}_f(t_1) \wedge forall.h(\lambda \boldsymbol{y}.\, \mathsf{ALL}_f(t_0),\, \lambda \boldsymbol{y}.\, t_0,\, t_1)$$
$$\mathsf{ALL}_f(\textit{if}\ t_1\ \textit{then}\ t_2\ \textit{else}\ t_3) := \mathsf{ALL}_f(t_1) \wedge \textit{if}\ t_1\ \textit{then}\ \mathsf{ALL}_f(t_2)\ \textit{else}\ \mathsf{ALL}_f(t_3)$$

for any variable v, any first-order function $g \neq if$, $g \neq f$, and any second-order procedure h (including proc). We write $\boldsymbol{y}$ as an abbreviation of $y_1, \ldots, y_k$, and $A \wedge B$ abbreviates "if A then B else false".

Example 1. Procedure $\mathit{forall.map}(p : @A \to \mathit{bool},\ f : @A \to @B,\ k : \mathit{list}[@A]) : \mathit{bool}$ returns *true* if and only if $p(z)$ is satisfied for all elements z of list k, because procedure *map* applies f to all elements z of k. ♢

Example 2. For the second-order procedure *foldl*,

$$\texttt{procedure}\ \mathit{forall.foldl}(p : @A \times @B \to \mathit{bool},\ f : @A \times @B \to @A,\\ x : @A,\ k : \mathit{list}[@B]) : \mathit{bool}$$

checks if $p(a, b)$ is satisfied for all pairs (a, b) that f is applied to by *foldl*. ♢

4.2 Generation of Verification Conditions

If a procedure f with a precondition is passed to a second-order procedure *proc*, we can use the quantification procedure *forall.proc* to find out if *proc* calls f only with arguments that satisfy the precondition of f:

Definition 4. *The* verification condition $VC(t) \in \mathcal{T}(\Sigma(P), \mathcal{V})_{bool}$ *for a given term* $t \in \mathcal{T}(\Sigma(P), \mathcal{V})$ *is defined by*

$$\begin{aligned}
&VC(x) := \mathit{true}\\
&VC(f(t_1, \ldots, t_n)) := c_f[t_1, \ldots, t_n] \wedge VC(t_1) \wedge \ldots \wedge VC(t_n)\\
&VC(h(\lambda \boldsymbol{y}.\, t_0,\ t_1)) := c_h[\lambda \boldsymbol{y}.\, t_0,\ t_1] \wedge VC(t_1) \wedge \mathit{forall.h}(\lambda \boldsymbol{y}.\ VC(t_0),\ \lambda \boldsymbol{y}.\, t_0,\ t_1)\\
&VC(\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3) := VC(t_1) \wedge \mathit{if}\ t_1\ \mathit{then}\ VC(t_2)\ \mathit{else}\ VC(t_3)\\
&VC(\lambda \boldsymbol{y}.\, t_0) := \mathit{true}
\end{aligned}$$

for any variable x, any first-order function $f \neq if$ with context requirement c_f, and any second-order procedure h with context requirement c_h.

The generation of verification conditions is similar to the synthesis of quantification procedures. However, there are some differences:

- Function f can be any first-order function here, while f refers to a particular parameter of *proc* in Definition 3.
- For calls of second-order procedures h, we need to check whether the context requirement of h is satisfied. Definition 3 does not contain such a check.
- Term t can be a λ-expression, so we have an additional clause for this case.

Example 3. For term $t := t_1/t_2$ we get $VC(t) = t_2 \neq 0 \wedge VC(t_1) \wedge VC(t_2)$. ♢

Example 4. For term $t := \mathit{map}(\lambda n : \mathbb{N}.\, c/n,\ l)$ we get

$$VC(t) = \mathit{forall.map}(\lambda n : \mathbb{N}.\, n \neq 0,\ \lambda n : \mathbb{N}.\, c/n,\ l)\,,$$

which expresses that $n \neq 0$ needs to hold for all values n that *map* calls $\lambda n : \mathbb{N}.\, c/n$ with. This is equivalent to $\forall n \in l.\ n \neq 0$ as desired. ♢

Definition 4 easily generalizes to second-order procedures h with more than two parameters:

Example 5. Suppose that a program contains a procedure

`procedure` $\mathit{ordered}(l : \mathit{list}[\mathbb{N}]) : \mathit{bool} <= \ldots$

that returns *true* if and only if list l is ordered and a procedure

`procedure` $\mathit{insert}(l : \mathit{list}[\mathbb{N}],\ n : \mathbb{N}) : \mathit{list}[\mathbb{N}] <=$
`assume` $\mathit{ordered}(l); \ldots$

that inserts a number n into an ordered list l. Then $\mathit{foldl}(\mathit{insert}, \emptyset, l)$ is an implementation of *insertion sort*. The verification condition for this term is $\mathit{forall.foldl}(\lambda k : \mathit{list}[\mathbb{N}], m : \mathbb{N}.\ \mathit{ordered}(k),\ \mathit{insert},\ \emptyset,\ l)$, which expresses that all lists k that occur as intermediate results need to be ordered. $\diamondsuit$

4.3 Discussion

When a procedure *proc* calls another procedure f or a function like *pred* or *tl*, it is the responsibility of the calling procedure *proc* to ensure that f is called with arguments that satisfy the precondition of f. For example, it is the responsibility of procedure *len* (cf. Fig. 1) to ensure that $\mathit{tl}(l)$ is only called if $l \neq \emptyset$.

However, if f is not a concrete function, but a function parameter of *proc*, then the precondition of f is unknown to *proc*, because f may be instantiated with quite arbitrary functions. Thus it becomes the responsibility of the *caller* of *proc* to instantiate f in such a way that all f-calls by *proc* meet the precondition of f. For instance, *map* should only be called with a function f and a list l such that f is applicable to all elements of l (cf. Fig. 4).

Using our approach, for each call $\mathit{proc}(f, \ldots)$ of a second-order procedure *proc* a verification condition can be generated that checks whether *proc* calls f only with arguments that satisfy the precondition of f. If one wanted to (or had to) specify this check explicitly as a precondition of *proc*, one could specify $\forall x \in l.\ c_f[x]$ as a precondition of *map*, for example. Even for procedures that are only slightly more complicated than *map*, this would soon become tedious: For instance, procedure *foldl* applies f to tuples (a, b), where $b \in l$ and $a : @A$ is "some intermediate result of $f(\ldots f(f(x, \mathit{hd}(l)), \mathit{hd}(\mathit{tl}(l))), \ldots)$". This is of course too informal to turn the observation into a precondition. Specifying a more general precondition for *foldl* such as $\forall a : @A.\ \forall b \in l.\ c_f[a, b]$ would be an easy way out of this imprecision, but it would be a stronger precondition than what *foldl* actually requires.

In this sense, our approach generates the *weakest* verification condition that ensures that all preconditions of functions are satisfied when a given term is evaluated. Developers only need to specify preconditions that are relevant from the algorithmic point of view (e.g., for procedure "!!" in Fig. 1). Bookkeeping-like preconditions that just propagate the preconditions of function parameters to the respective second-order procedure can be omitted, because our approach analyzes these indirect function calls automatically.

```
structure $pair[@A, @B]$ <= $mkpair(fst : @A,\ snd : @B)$
structure $variable.symbol$ <= $variable(varID : \mathbb{N})$
structure $function.symbol$ <= $func(funcID : \mathbb{N})$
structure $term$ <=
  $var(vsym : variable.symbol)$,
  $apply(fsym : function.symbol,\ args : list[term])$

procedure $lookup(key : @A,\ alist : list[pair[@A, @B]]) : @B$ <=
if $alist = \emptyset$
  then $\star$
  else if $key = fst(hd(alist))$
         then $snd(hd(alist))$
         else $lookup(key, tl(alist))$

procedure $sig.known(t : term,\ arity : list[pair[function.symbol, \mathbb{N}]]) : bool$ <=
if $?var(t)$
  then $true$
  else if $\nabla_{lookup}(fsym(t), arity)$
         then $every(\lambda s : term.\ sig.known(s, arity),\ args(t))$
         else $false$

procedure $wellformed(t : term,\ arity : list[pair[function.symbol, \mathbb{N}]]) : bool$ <=
assume $sig.known(t, arity)$;
if $?var(t)$
  then $true$
  else if $len(args(t)) = lookup(fsym(t), arity)$
         then $every(\lambda s : term.\ wellformed(s, arity),\ args(t))$
         else $false$
```

Fig. 5. A functional program that checks if a given term is well-formed

4.4 Example

The program in Fig. 5 uses both of the concepts to implement procedures with preconditions that we described in Sect. 2. Furthermore, it contains both direct and indirect procedure calls that illustrate the generation of the corresponding verification conditions.

Procedure *lookup* returns the second component b of the first pair $mkpair(a, b)$ in list *alist* such that $a = key$. This procedure is incompletely defined, because there might be no such pair in *alist*. The corresponding domain procedure ∇_{lookup} thus returns *true* if and only if *alist* contains a pair $mkpair(a, b)$ with $a = key$. Hence the implicit precondition of *lookup* is given by $\nabla_{lookup}(key, alist)$.

Procedure *sig.known* checks whether the signature of all function symbols occurring in a given term t is known, i.e., whether *arity* defines the arity for each function symbol f in t. This procedure is used in the context requirement of procedure *wellformed*. Given a term t and a list *arity* that defines the arity of each function symbol occurring in t, procedure *wellformed* returns *true* if and only if the number of arguments of each function application in term t is equal to the arity of the leading function symbol.

For the direct call of procedure *lookup* in the body of *wellformed*, we get the verification condition $\nabla_{lookup}(fsym(t), arity)$. Since we may assume that $sig.known(t, arity)$ holds, this verification condition is true. For the indirect call of procedure *wellformed* via the second-order procedure *every*, we get the verification condition

$$\begin{aligned}&forall.every(\lambda s : term.\ sig.known(s, arity),\\&\qquad\qquad \lambda s : term.\ wellformed(s, arity),\\&\qquad\qquad args(t))\ .\end{aligned}$$

This requires $sig.known(s, arity)$ to hold for each element s of list $args(t)$ (i.e., each subterm s of t) that is passed to *wellformed*. Since $sig.known(t, arity)$ by definition entails $sig.known(s, arity)$ for all subterms s of t, this verification condition is true as well and can easily be verified in √eriFun.

5 Related Work

Annotating programs with preconditions has a long history and is supported (in various ways) in several programming languages such as C, Eiffel, Haskell, and Java. In the following, we mainly consider related work that concerns programs with preconditions in the context of semi-automated theorem proving.

In ACL2 [8], procedures can be annotated by *guards* to specify preconditions. By *guard verification*, theorems are proved that ensure that a procedure satisfies the guards of all procedures that it uses in its body. Using and verifying guards is optional in ACL2, and all procedures need to be completely defined in case that guard checking mode is turned off. ACL2 has a first-order programming language, so second-order procedures such as *map* or *foldl* cannot be defined in a way that would allow procedure calls like $map(f, l)$ for concrete functions f [5,7].

In PVS [11], one can express preconditions by using specific types. For example, for the second parameter of a division algorithm on natural numbers one can use the type $\{n : \mathbb{N} \mid n \neq 0\}$. Preconditions that involve more than a single parameter can be expressed using dependent types. For instance, the type for parameter l of procedure "!!" (cf. Fig. 1) can be specified by $\{k : list[@A] \mid len(k) > n\}$. PVS generates *type-correctness conditions* to ensure that the arguments of procedure calls have the required types. Thus PVS uses type checking to verify that all preconditions are satisfied, while our approach can also be used in languages that do not support dependent types.

In Isabelle/HOL [10], developers of theories usually try to define functions "as completely as possible". For instance, *division* is completed by defining $n/0 := 0$ and function *tl* is completed by defining $tl(\emptyset) := \emptyset$. If there is no way of completing the definition of a function, the result can be left unspecified for some cases; e.g., function *hd* is defined just by $hd(x :: xs) := x$, so $hd(\emptyset)$ is a value (because all functions are total) that nothing is known about except its type (because no defining equation is applicable). This corresponds to √eriFun's concept of incompletely defined procedures [19] in its original form (i.e., without our reinterpretation as procedures with implicit preconditions).

For the programming language Haskell [12], Xu et al. present an approach to statically check contracts in programs [20]. Similarly to PVS, contracts are expressed by dependent types. Differently from PVS, ACL2, and √eriFun, however, the verification part is not tackled by a general purpose theorem prover. Instead, the term under consideration is uniformly transformed by wrapping function calls with checks that make the evaluation "crash" when a contract is violated. Then a dedicated symbolic evaluator tries to rewrite the transformed term until it is syntactically "crash-free" (i.e., it does not contain "crash" commands anymore). The symbolic evaluator is tailored to the lazy evaluation strategy of Haskell. A prototype implementation is reported to be able to "prove some simple contract satisfaction checks, but is still incomplete for more sophisticated examples involving the use of recursive functions in predicates" [20].

Regarding imperative programming languages, Eiffel [9] checks at run-time if contracts are satisfied, while Spec# [4] in addition supports static verification using the automatic verifier Boogie. According to [20], contracts that involve recursive procedures cannot be analyzed statically in Spec#. In our approach, recursive procedures pose no problem, because √eriFun is specifically designed to prove properties about recursive procedures.

As an alternative to the semantics that we considered in this paper, Sabel and Schmidt-Schauß [13] present a contextual semantics that equates undefinedness with non-termination. Thus division by zero is regarded like a non-terminating evaluation. The static analysis whether all procedure calls in a program satisfy the respective preconditions is not considered in [13].

6 Conclusion

In order to ensure that the execution of a program will never fail due to violated preconditions, one can statically analyze if all function calls in the program are safe in the sense that the arguments of each function call satisfy the precondition of the respective function. In this paper, we proposed a method to statically analyze function calls in second-order functional programs.

Our method generates verification conditions for function calls in a given term. In particular, our approach works for programming languages (and specification languages) that do not offer dependent types and the corresponding machinery for type checking. It is able to handle indirect function calls, e.g., calls of function f in $map(f, l)$, that occur when a function is passed as an argument to a second-order procedure. By analyzing the behavior of the second-order procedure with respect to its function parameter, our approach generates the weakest verification condition that ensures that the indirect function calls satisfy the precondition of the function.

We offer developers two possibilities to conveniently implement procedures with preconditions: Firstly, preconditions can be specified explicitly as a so-called context requirement of the procedure [14]. In this context requirement, any (terminating) procedure of the program can be used to specify the precondition. Secondly, preconditions can be specified implicitly by defining a procedure incompletely. In this case, a domain procedure is synthesized using the approach by Walther and Schweitzer [19] that characterizes the inputs of the procedure that do not lead to an undefined case.

In [19], the motivation for incompletely defined procedures was to model procedures with run-time exceptions in a way that does not mix up non-termination and run-time exceptions in a single notion of partiality. Thus reasoning techniques for *total* functions can be reused to reason about procedures with run-time exceptions. The absence of exceptions did *not* have to be proved, so one frequently had to reason about indetermined values. In our approach, we consider incompletely defined procedures as just another convenient way of implementing procedures with preconditions. This has the advantage that we can give a simpler definition of the semantics of first-order and second-order programs compared to [19], because the verification conditions (once proved) guarantee the absence of exceptions, so the semantics need not model indetermined values anymore.

We restricted our consideration to second-order programs to keep the semantics simple. For instance, if a call of procedure *lookup* (cf. Fig. 5) could instantiate type variable @A with a function type, then the semantics would need to define when two functions are to be considered as equal. However, equality of functions is undecidable in general. By allowing the instantiation of type variables with base types only, we avoid this problem and still get a programming language where we can investigate the challenges of indirect function calls.

Our approach has been implemented in an experimental version of √eriFun. This gives us the full power of a semi-automated inductive theorem prover to verify the generated verification conditions. The proofs of verification conditions are typically relatively straightforward; often a single induction suffices to reason about recursively defined procedures that occur in the preconditions. In summary, we think that our approach effectively helps to find program errors due to violated preconditions early so that one does not need to worry about indetermined behavior once the verification conditions have been verified.

Acknowledgment. I would like to thank Christoph Walther for many discussions about incompletely defined procedures and procedures with preconditions as well as for his support in integrating the method described in this paper into √eriFun. Furthermore, I am grateful to Moritz Sinn and to Nathan Wasser for implementing the approach. I thank Simon Siegler for detailed feedback on a draft of this paper. Alexander Lux and Artem Starostin provided helpful comments on parts of this paper.

References

1. M. Aderhold. *Verification of Second-Order Functional Programs.* Doctoral dissertation, TU Darmstadt, 2009.
2. M. Aderhold. Automated synthesis of induction axioms for programs with second-order recursion. In J. Giesl and R. Hähnle, editors, *Proceedings of IJCAR-5*, volume 6173 of *LNCS*, pages 263–277. Springer, 2010.
3. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Kluwer Academic Publishers, 2002.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
5. R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In Vladimir Lifschitz, editor, *Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
6. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 366–373. Springer, 1995.
7. W. A. Hunt Jr., M. Kaufmann, R. B. Krug, J S. Moore, and E. W. Smith. Meta reasoning in ACL2. In J. Hurd and T. F. Melham, editors, *Proceedings of TPHOLS-2005*, volume 3603 of *LNCS*, pages 163–178. Springer, 2005.
8. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
9. B. Meyer. *Eiffel: The Language.* Prentice Hall International, London, 1992.
10. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Springer, June 2010.
11. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, November 2001.
12. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
13. D. Sabel and M. Schmidt-Schauß. Reconstruction of a logic for inductive proofs of properties of functional programs. Frank report 39, J. W. Goethe-Universität, Frankfurt am Main, Germany, June 2010.
14. A. Schlosser, C. Walther, M. Gonder, and M. Aderhold. Context dependent procedures and computed types in √eriFun. In *Proc. of 1st Workshop Programming Languages meet Program Verification*, volume 174 of *ENTCS*, pages 61–78, 2007.
15. C. Walther. *Semantik und Programmverifikation.* Teubner-Wiley, Leipzig, 2001.
16. C. Walther, M. Aderhold, and A. Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, TU Darmstadt, 2006.
17. C. Walther and S. Schweitzer. About √eriFun. In F. Baader, editor, *Proc. of CADE-19*, volume 2741 of *LNCS*, pages 322–327. Springer, 2003.
18. C. Walther and S. Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.
19. C. Walther and S. Schweitzer. Reasoning about incompletely defined programs. In G. Sutcliffe and A. Voronkov, editors, *Proc. of LPAR-12*, volume 3835 of *LNAI*, pages 427–442. Springer, 2005.
20. D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 41–52. ACM, 2009.