Application and Formal Specification of Sorted Term-Position Algebras

Arnd Poetzsch-Heffter and Nicole Rauch

University of Kaiserslautern
{poetzsch, rauch}@informatik.uni-kl.de

Abstract. Sorted term-position algebras are an extension of term algebras. In addition to sorted terms with constructor and selector functions, they provide term positions as algebra elements and functions that relate term positions. This paper describes possible applications of term-position algebras and investigates their formal specification in existing specification frameworks. In particular, it presents an algebraic specification of term-positions in CASL and in a higher-order logic.

1 Introduction

Sorted term algebras are a very helpful and flexible concept for modeling and programming. In particular, they provide the foundation for the datatype declarations in functional programming languages and sorted specification languages (see e.g. [1, 2]). *Term-position algebras*, or *tepos-algebras* for short, are an extension of term algebras. Conceptually, a term position is a node within a given sorted tree. While for a constructor term it only makes sense to ask for its subterms, term positions enable to refer to parent positions and, more generally, to the upper tree context of positions. Formally, a term position p in a constructor term t is the occurrence of a subterm s of t in t. We call s the *term belonging to* p and t the *root term* of p. The tepos-algebra for a given sorted term algebra $\mathcal{A}$ and a sort S of $\mathcal{A}$ is an extension of $\mathcal{A}$ by all positions in constructor terms of sort S.

An important aspect for the practical use of term-position algebras is that they need no further declaration constructs and almost no additional declaration work by the user[1]. They are defined based on the usual language constructs for datatype declaration. In this paper, we investigate the design and the formal specification of the semantics of sorted tepos-algebras. The goal is to use existing specification and verification frameworks for the semantics specification so that their tooling and verification support can be exploited. As specification frameworks, we consider CASL [3–5] and Isabelle/HOL [6]. The contribution of the paper has different aspects: It introduces tepos-algebras as a powerful language concept and their formalization as an interesting specification challenge. In the main parts of the paper, we describe how this challenge can be solved in CASL and Isabelle/HOL and compare the two specifications.

[1] By a *user*, we mean a person who writes programs or specifications based on term and tepos-algebras.

J.L. Fiadeiro, P. Mosses, and F. Orejas (Eds.): WADT 2004, LNCS 3423, pp. 201–217, 2005.

Overview. The rest of the paper is structured as follows. Section 2 provides an informal introduction to the use of tepos-algebras by a small example. Section 3 explains the design choices underlying the specification of tepos-algebras and formulates the specification challenge. Section 4 presents the specification of tepos-algebras in CASL. Section 5 shows how tepos-algebras can be specified in Isabelle/HOL. Section 6 discusses the approach in relation to other work. Section 7 contains the conclusions.

2 Tepos-Algebras at Work

In this section, we show how tepos-algebras can be used in programming and specification. With this introduction, we pursue four goals:

- The reader should get some intuitive understanding of how tepos-algebras can be applied. According to our experiences[2], working with constructor terms and term positions, that is, with two tree representations at once, is unfamiliar at the beginning, but well accepted after having studied some examples.
- We want to give some idea of how tepos-algebras can be integrated into programming or specification languages.
- To motivate the study of tepos-algebras, we like to demonstrate that they enable new specification techniques. In the example below, we show two such aspects from the area of programming language specification: 1. Simplifying the formulation of context conditions. 2. Avoiding continuation semantics for a language with gotos.
- A subset of the example will later be used to illustrate the formal specification of tepos-algebras.

For illustration purposes, we assume a fictitious programming or specification language TePos with a datatype construct for the declaration of free recursive datatypes with constructors and selectors (such datatype declarations are available in most typed functional programming languages and specification languages).

Datatype Declaration. In TePos, the declaration of the abstract syntax of a small imperative programming language with gotos is as follows:

```
datatype SIMPL is
  Prog = prgm( stm: Stmt )
  Stmt = assg( lhs: Idt , rhs: Expr )
       | sequ( fst: Stmt, scd: Stmt )
       | loop( cnd: Expr, bod: Stmt )
       | goto( tid: Idt )
       | labl( lid: Idt,  stm: Stmt )
  Expr = vare( idt: Idt )
       | cons( val: Int )
       | plus( fst: Expr, scd: Expr )
end
```

[2] Most of our experiences were made with students in compiler construction courses, in which we used a tool based on tepos-algebras [7, 8].

This declaration uses the sorts `Idt` for identifiers and `Int` for integer constants, it introduces SIMPL[3] as a name for the declaration, and defines new sorts `Prog` with constructor `prgm` as well as `Stmt` and `Expr` with constructors for the different statement and expression kinds. Besides term sorts and constructors, it provides term selectors like `stm`, `lhs`, and `rhs` that allow to select the subterm of a given term. Selectors are partial functions. For example, the evaluation of `fst( assg("a", cons(8)) )` is not defined, because `fst` is a selector that only works for terms constructed by `sequ`. How partiality is handled in TePos is irrelevant for this paper. We allow overloading of selector names if their domain sorts are different. Otherwise overloading is not allowed.

Tepos-Algebra Declaration. TePos supports a declaration that provides the elements and features of a tepos-algebra. The tepos-algebra is defined as an extension of a datatype (here `SIMPL`) and one of its sorts (here `Prog`). As a third argument, it takes a string (here `"Pos"`) that is used to name position sorts. Here is the declaration for our example:

```
datatype SIMPLPOS is tepos of SIMPL, Prog, "Pos" end
```

This one-line declaration defines the tepos-algebra with a number of sorts and functions. It defines the sorts `ProgPos`, `StmtPos`, `ExprPos`, `IdtPos`, and `IntPos` of positions in terms of sort `Prog`. For example, an element of sort `StmtPos` represents a subterm occurrence of sort `Stmt` in a term of sort `Prog`. The declaration also defines the overloaded functions

```
term: ProgPos -> Prog          pos:  Prog    -> ProgPos
term: StmtPos -> Stmt          root: StmtPos -> ProgPos
term: ExprPos -> Expr          root: ExprPos -> ProgPos
term: IdtPos  -> Idt           root: IdtPos  -> ProgPos
term: IntPos  -> Int           root: IntPos  -> ProgPos
```

The function `term` yields the term belonging to a position (as defined in Sect. 1); `pos` yields the root position of a term of sort `Prog`; and `root` yields the root position for a given position. Thus, `root` is a first example of a function on positions p that refers to the upper tree context of p.

To reach child positions, that is, positions down the tree, the declaration `SIMPLPOS` defines selectors for positions. To keep the naming simple, we overload the term selectors. For example, the selector `cnd: Stmt -> Expr` is overloaded by a selector `cnd: StmtPos -> ExprPos`. Both selectors are partial functions, and the position selector is defined for a position p if and only if the term selector is defined for the term belonging to p. Altogether, we get two tree representations linked by the functions `pos` and `term`. Figure 1 illustrates this for a simple term.

By distinguishing between datatype constructors and other functions, Figure 1 also indicates a central aspect of how tepos-algebras are specified. Argument flow of datatype constructors is denoted by solid arrows. For the other

[3] **S**imple **I**mperative **P**rogramming **L**anguage.

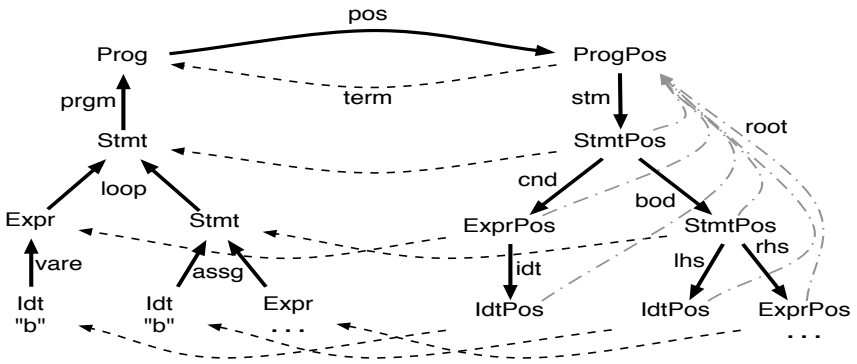

Fig. 1. Illustration of tree representations with terms and positions

functions, we use dashed arrows. Note that the position tree is constructed from the root to the leaves (see Sect. 4).

Tepos-Algebra Extension and Use. A fully-fledged language supporting tepos-algebras would provide further features. In this section, we illustrate and use subsorting on term and position sorts and an extended form of pattern matching. In Section 4, we show how a supersort of all position sorts can be specified and used as a basis for further functions.

Subsorting for free datatypes is naturally defined by the constructors. All terms constructed by a constructor c form one subsort of the range sort of c. We denote the subsorts by the constructor name with a capitalized first letter. For example, `Goto` denotes the goto-statements, that is, the subsort of `Stmt` that contains exactly those terms constructed by `goto`. The corresponding subsorts on positions are denoted by the postfix "Pos" (for example, `GotoPos`). It should be clear that such a subsorting needs no special declarations by the user but can be implicitly provided by the language used.

Based on the declarations `SIMPL` and `SIMPLPOS`, we can define interesting language properties in an elegant declarative way. We start with context conditions. For example, labels must be unique in SIMPL-programs. That is, two different labeled statements *lp1, lp2* in the same program ($\mathrm{root}(lp1) = \mathrm{root}(lp2)$) must have different label identifiers:

$$\forall\ \texttt{LablPos}\ lp1, lp2 :\\ lp1 \neq lp2\ \wedge\ \texttt{root}(lp1) = \texttt{root}(lp2)\ \Rightarrow\ \texttt{term}(\texttt{lid}(lp1)) \neq \texttt{term}(\texttt{lid}(lp2))$$

Recall that applying the selector `lid` to a labeled statement position yields an identifier position. To get the identifier at that position, we have to apply the function `term` (cf. Fig. 1). The second context condition states that for each goto statement there must be a corresponding labeled statement:

$$\forall\ \texttt{GotoPos}\ gp\ \ \exists\ \texttt{LablPos}\ lp :\\ \texttt{root}(gp) = \texttt{root}(lp)\ \wedge\ \texttt{term}(\texttt{tid}(gp)) = \texttt{term}(\texttt{lid}(lp))$$

It is worth noting that, without positions, these properties can only be formalized by a nontrivial environment or symboltable mechanism.

For SIMPL-programs satisfying the context conditions, we can define a function `target` that yields for a goto statement the unique target statement:

```
target:  GotoPos ->  LablPos
```

The specification of `target` is given in the appendix. Here, it is of interest that the user of `target` need not know about the specification details. The function `target` links one position of the tree to another one. In particular, we can use it to express an operational semantics for SIMPL without continuations (see [9] for a discussion on continuation semantics). We present such a semantics here as an example to discuss pattern matching on positions. Let `State` be the sort of mappings from identifiers to integers, `eval` be a function evaluating an expression in a state, and `update` be a function that takes a state *st*, an identifier *id*, and a value *v* and yields a "new" state *nst* such that $nst(i) = st(i)$ for all $i \neq id$ and $nst(id) = v$:

```
State  =  Idt -> Int              update :  State x Idt x Int -> State
eval   :  Exp x State -> Int
```

Based on these notions, the execution of a SIMPL-program *p* in state *st* is defined by `exec(stm(pos(`p`)),`st`)` where `exec` is specified as follows:

```
exec:  StmtPos x State -> State
exec(sp, st) = case sp of
   assg<v,e>     =>  update(st,term(v),eval(term(e),st))
 | sequ<sp1,sp2> =>  exec(sp2, exec(sp1,st))
 | loop<e,bod>   =>  if eval(term(e),st)=0  then st
                        else exec(sp,exec(bod,st))
 | Goto<_>       =>  exec(target(sp),st)
 | Labl<_,sp0>   =>  exec(sp0,st)
```

The case expression is similar to that of functional programming languages. The difference is that matching works on positions. For example, the pattern `assg<v,e>` matches statement positions of sort `AssgPos` with child positions *v* and *e*. The reason to use a position instead of a term representation of statements is that the execution of goto statements refers to the target statement in the upper context. This can not directly be expressed by constructor terms.

This section should have given some idea of how tepos-algebras can be used in programming and specification. Further examples as well as language and implementation issues are described in [7]. The following sections focus on the challenge of how tepos-algebras can be formally specified.

3 Specification Challenge

On the meta-level, term positions are usually formalized as pairs with the root term as first component and a sequence of natural numbers as second component.

The number sequence describes the selection path from the root position of the term to the subterm position. To illustrate this, let t be the term of Fig. 1:

```
prgm( loop( vare("b"), assg("b",plus(vare("b"),vare("c"))) ) )
```

Using brackets to enclose list elements in the meta-notation, the position at the root of the program is denoted by $(t, [])$, the position of the loop statement by $(t, [1])$, of the assignment by $(t, [1, 2])$ and of the identifier "b" on the left hand side of the assignment by $(t, [1, 2, 1])$. For object-level specifications, this approach has the following four disadvantages: (a) Positions are not sorted; (b) selection by numbers is error-prone; (c) modifications or extensions of the term algebra (e.g. adding a parameter to a constructor) cause subtle modifications of the position handling; (d) the algebraic laws of term positions are hidden. To overcome these disadvantages, tepos-algebras should be formalized within a specification framework in a way that positions are ordinary sorted elements.

The main design problem for tepos-algebras pertains to the sorting/typing discipline for the positions. Essentially, there are four options:

1. All positions of all terms are in one sort.
2. Positions are sorted according to the term sorts they correspond to. That is, there is exactly one position sort for each term sort.
3. In addition to the second option, position sorts are distinguished with respect to the *sort* of the root term. That is, a position sort captures the information about the sort of the root.
4. Position sorts are dependent sorts, depending on the root term.

For the following reasons, we chose the third design option: It is sufficiently fine grained for the applications that we are interested in and that we can imagine so far (see Sect. 2 and [7]). The more coarse grained sorts of the first and second option can be realized within this option by introducing further supersorts. We avoid dependent sorts that are not supported by many specification frameworks. Based on this design decision, the *specification challenge* is as follows:

> Given a sorted free datatype specification with sorts $S_0, ..., S_n$, suitable constructors and selectors, and a sort $S \in \{S_0, ..., S_n\}$, specify the corresponding tepos-algebra with suitable sorts and functions.

Essentially, there exist two approaches to formalize new language concepts or constructs. Either one writes a freestyle mathematical definition, or one uses existing specification languages and frameworks. The first approach provides more flexibility, the second approach allows to inherit the techniques and tools underlying the specification framework. Here, we investigate the second approach. As specification frameworks, we use the algebraic order-sorted specification language CASL and the higher-order many-sorted specification language of Isabelle/HOL. For both frameworks, we specify tepos-algebras by a shallow embedding, that is, we define how a tepos-algebra declaration like that for SIMPLPOS given above is translated into the specification language.

4 Specifying Tepos-Algebras in an Algebraic Framework

The specification of tepos-algebras in CASL is described in two steps. In the first step, we concentrate on the kernel of tepos-algebras containing the position sorts, the selectors on position sorts, and the functions `pos` and `term`. As an introduction, we demonstrate the shallow embedding of the kernel by a representative example (Sect. 4.1). Then, we define it for the general case (Sect. 4.2). In the second step, we explain how extensions of the kernel can be formalized in CASL (Sect. 4.3).

4.1 Introduction to the Tepos-Algebra Specification in CASL

The declaration of a tepos-algebra consists of three parts:

1. a declaration of a free datatype,
2. a declaration of the sort of terms for which the positions should be defined,
3. declarations for the naming of new sorts and functions.

In CASL, the free datatype can be given as a named specification based on some externally declared sorts. As a tiny example, we consider a subset of the abstract syntax of SIMPL (cf. Sect. 2). The extension to SIMPL is straightforward. In CASL syntax, we get the following declaration:

```
spec SIMPLS = sort Idt  then free types
    Prog ::= prgm( stm:? Stmt );
    Stmt ::= assg( lhs:? Idt ; rhs:? Expr )
           | sequ( fst:? Stmt; scd:? Stmt )
           | loop( cnd:? Expr; bod:? Stmt );
    Expr ::= vare( idt:? Idt )
           | plus( fst:? Expr; scd:? Expr )
```

The question mark after the selector names indicates that selectors are partial functions. Note that CASL allows overloading of functions as demonstrated by the selector `fst`. To declare the tepos-algebra for SIMPLS, we could imagine an extension of CASL allowing declarations like:

```
spec SIMPLSPOS = tepos(SIMPLS,Prog,"Pos")
```

The meaning of this declaration is defined by giving a CASL specification for it. The *basic idea* underlying this specification is taken from the meta-level representation of a position as a pair of the root term and a list of natural numbers describing the selection path – recall the example $(t, [1, 2, 1])$ from above. To express the position at the root, we use a constructor `pos`, that is, we write $\texttt{pos}(t)$ instead of $(t, [])$. The selection of child positions is denoted by unary functions as well. For convenience, we reuse the names of the selectors on the term side for these functions. For example, $(t, [1, 2, 1])$ would be denoted on the object-level as $\texttt{lhs}(\texttt{bod}(\texttt{stm}(\texttt{pos}(t))))$. This overloading can be handled by CASL if position sorts are different from term sorts and if position sorts corresponding to different term sorts are different as well. Our approach fulfills this requirement; recall our design decision described in the previous section. Following this basic idea leads to two specification problems:

1. How do we specify that different selection paths yield different positions?
2. How do we distinguish "valid" selection paths from "invalid" ones, that is, from paths that do not denote a position in the root term?

The first question has a canonical answer: Use a free type specification in which the selection functions `stm`, `bod`, etc. are the constructors of the position sorts. Unfortunately, this leads to a conflict with the second problem, because we get many invalid paths. To overcome this conflict, we can use partial constructors and enforce that they are defined if and only if the path is valid. As a partial function yields "undefined" in a free specification whenever we do not force it explicitly to yield a defined value, we only have to specify in which cases the paths are valid.

A path is valid iff all selection steps are valid. A selection step by selection function *sel* on a position *pp* is valid iff the selection by *sel* is defined on the term belonging to *pp*. To formalize this, we have to specify a function `term` that yields for each position the term belonging to it. `term` can be defined recursively: For the root position of a term p, we have $\texttt{term}(\texttt{pos}(p)) = p$. Otherwise, if *pp* is a position and *sel* is a selection function for the sort of *pp*, then $\texttt{term}(sel(pp)) = sel(\texttt{term}(pp))$.

The main challenge now is that the specification of the partial constructors and the recursive specification of `term` are mutually dependent. Thus, in order to implement these ideas in a specification framework, it has to support free specifications of this kind for types with partial constructors and for total recursive functions. CASL meets this challenge. Thus, our specification approach can directly be formulated in CASL. Figure 2 demonstrates this for `SIMPLSPOS`.

The next subsection provides a complete description of the embedding that we illustrated here by the example.

4.2 Complete Description of the Embedding

In this subsection, we describe how the tepos-algebra for a given datatype declaration is specified in general. Furthermore, we discuss validation issues. Tepos-algebras are declared based on datatype declarations of the following form:

$$
\begin{array}{l}
\texttt{spec DT} = \texttt{sorts } \mathrm{U}_1, \ldots, \mathrm{U}_p \texttt{ then free types} \\
\quad \mathrm{S}_1 ::= con^1_1 \quad (sl^1_{1,1} :? \mathrm{T}^1_{1,1} ; \ldots ; sl^1_{1,n(1,1)} :? \mathrm{T}^1_{1,n(1,1)}) \\
\qquad \ldots \\
\qquad\ | \quad con^1_{m(1)} (sl^1_{m(1),1} :? \mathrm{T}^1_{m(1),1} ; \ldots ; sl^1_{m(1),n(1,m(1))} :? \mathrm{T}^1_{m(1),n(1,m(1))}); \\
\quad \ldots \\
\quad \mathrm{S}_r ::= con^r_1 \quad (sl^r_{1,1} :? \mathrm{T}^r_{1,1} ; \ldots ; sl^r_{1,n(r,1)} :? \mathrm{T}^r_{1,n(r,1)}) \\
\qquad \ldots \\
\qquad\ | \quad con^r_{m(r)} (sl^r_{m(r),1} :? \mathrm{T}^r_{m(r),1} ; \ldots ; sl^r_{m(r),n(r,m(r))} :? \mathrm{T}^r_{m(r),n(r,m(r))})
\end{array}
$$

where S_i are different sort names and $\mathrm{T}^i_{j,k}$ denote sorts that are either in the *defined sorts* $\{\mathrm{S}_1, \ldots, \mathrm{S}_r\}$ or in the *used sorts* $\{\mathrm{U}_1, \ldots, \mathrm{U}_p\}$. We assume that the

```
spec SIMPLSPOS = SIMPLS then free {
  types
    ProgPos ::= pos( Prog );
    StmtPos ::= stm( ProgPos )?
              | fst( StmtPos )?
              | scd( StmtPos )?
              | bod( StmtPos )?;
    ExprPos ::= rhs( StmtPos )?
              | cnd( StmtPos )?
              | fst( ExprPos )?
              | scd( ExprPos )?;
    IdtPos  ::= lhs( StmtPos )?
              | idt( ExprPos )?;
  ops
    term : ProgPos -> Prog;
    term : StmtPos -> Stmt;
    term : ExprPos -> Expr;
    term : IdtPos  -> Idt;

  vars p: Prog; pp: ProgPos;
       sp: StmtPos; ep: ExprPos;
  . term(pos(p))  = p
  . term(stm(pp)) = stm(term(pp))
  . term(fst(sp)) = fst(term(sp))
  . term(scd(sp)) = scd(term(sp))
  . term(bod(sp)) = bod(term(sp))
  . term(rhs(sp)) = rhs(term(sp))
  . term(cnd(sp)) = cnd(term(sp))
  . term(fst(ep)) = fst(term(ep))
  . term(scd(ep)) = scd(term(ep))
  . term(lhs(sp)) = lhs(term(sp))
  . term(idt(ep)) = idt(term(ep))
}
```

Fig. 2. CASL specification for `SIMPLSPOS`

specification does not use the names `pos` and `term`, that all constructor names con^i_j are different, and that selectors are only overloaded if they have different domain sorts, that is, selector names $sl^{i_1}_{j_1,k_1}$ and $sl^{i_2}_{j_2,k_2}$ may only be equal if $i_1 \neq i_2$. To keep the following construction simple, we assume that there is at least one ground term for each used and defined sort. We say that a string π is an *admissible postfix* for a set $\mathcal{T}$ of sort names if no sort name in $\mathcal{T}$ ends with π. For brevity, we will not distinguish between sorts and their names in the following.

The declaration of a tepos-algebra for a datatype declaration DT with defined sorts $\mathcal{S}$ and used sorts $\mathcal{U}$ consists of a sort S in $\mathcal{S}$ and a postfix π admissible for $\mathcal{S} \cup \mathcal{U}$. To formalize the meaning of such a declaration, we need some notions

and notations. We say that T is a sort *reachable* from S iff there is a term of sort S with a subterm of sort T. In particular, S is reachable from S (recall that there is a term in S). The set of sorts reachable from S in DT is denoted by $\mathcal{R} = \{\mathrm{R}_1, \ldots, \mathrm{R}_q\}$. Without loss of generality, we assume that S equals R_1. Furthermore, we need new schematic names:

- The selector names with range type R_i and a domain type in $\mathcal{R}$ are denoted by $slr^i_1, \ldots, slr^i_{l(i)}$. Note that each schematic name slr^i_j denotes the same name as one of the schematic names $sl^k_{l,m}$.
- The index of the domain type of selector slr^i_j is denoted by $dom(i,j)$, that is, the domain type is $\mathrm{R}_{dom(i,j)}$.
- R^π_{ix} denotes the sort name obtained from R_{ix} by appending π where ix is a single or double index.

Based on these notations, the tepos-algebra for DT, R_1, and π is defined by the CASL specification shown in Fig. 3.

Validation. As the specification given in Fig. 2 *defines* the meaning of the tepos-algebra for datatype DT, it can only be validated and not verified. Validation

DT **then free** {

R^π_1 ::= **pos** (R_1)
| slr^1_1 ($\mathrm{R}^\pi_{dom(1,1)}$)?
...
| $slr^1_{l(1)}$ ($\mathrm{R}^\pi_{dom(1,l(1))}$)?;
R^π_2 ::= slr^2_1 ($\mathrm{R}^\pi_{dom(2,1)}$)?
...
R^π_q ::= slr^q_1 ($\mathrm{R}^\pi_{dom(q,1)}$)?
...
| $slr^q_{l(q)}$ ($\mathrm{R}^\pi_{dom(q,l(q))}$)?;

ops
term : $\mathrm{R}^\pi_1 \to \mathrm{R}_1$;
...
term : $\mathrm{R}^\pi_q \to \mathrm{R}_q$;

vars $x : R_1$; $x_1 : R^\pi_1$; ...; $x_q : R^\pi_q$;

$\cdot$ **term**(**pos**(x)) $= x$
$\cdot$ **term**($slr^1_1(x_{dom(1,1)})$) $= slr^1_1($ **term**$(x_{dom(1,1)})$)
...
$\cdot$ **term**($slr^1_{l(1)}(x_{dom(1,l(1))})$) $= slr^1_{l(1)}($ **term**$(x_{dom(1,l(1))})$)
...
$\cdot$ **term**($slr^q_1(x_{dom(q,1)})$) $= slr^q_1($ **term**$(x_{dom(q,1)})$)
...
$\cdot$ **term**($slr^q_{l(q)}(x_{dom(q,l(q))})$) $= slr^q_{l(q)}($ **term**$(x_{dom(q,l(q))})$)

}

Fig. 3. Complete embedding schema for tepos-algebras

here means to check that the specification formalizes our informal understanding and that it has the properties we expect (see [10] for a discussion). An essential property is for example that the extension exists and is unique (up to isomorphism). This holds because we used a free construction based on equational axioms only.

A second important validation property is that the elements in the position sorts represent exactly the positions in the terms of sort R_1. To show this and to illustrate where the CASL semantics comes in, let us assume that $\mathcal{A}$ is a partial algebra satisfying the specification. In the following, we consider all terms to be interpreted in $\mathcal{A}$. We first prove an auxiliary lemma. Then, we come back to the validation property.

Lemma 1. Let t be a term of sort R_1, let $sl_1, \ldots, sl_n$ be some selectors, and let $sl_n(\ldots sl_1(t)\ldots)$ be well-sorted. Then $sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)$ is well-sorted and:

1. $sl_n(\ldots sl_1(t)\ldots) \;=\; \mathtt{term}(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots))$ (strong equality)
2. $sl_n(\ldots sl_1(t)\ldots)$ is defined $\;\Leftrightarrow\;$ $sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)$ is defined

Proof of lemma 1: $sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)$ is well-sorted according to the construction of the specification. The first property is proved by induction on n. For $n = 0$, we get $\mathtt{term}(\mathtt{pos}(t)) = t$ as a direct consequence of the first axiom. Now, let us assume $sl_n(\ldots sl_1(t)\ldots) = \mathtt{term}(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots))$ and let sl be a constructor such that $sl(sl_n(\ldots sl_1(t)\ldots))$ is well-sorted. We derive:

$$
\begin{aligned}
& sl(sl_n(\ldots sl_1(t)\ldots)) \\
= \;& (* \text{ by induction hypothesis } *) \\
& sl(\mathtt{term}(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots))) \\
= \;& (* \text{ by the axiom corresponding to } sl\ *) \\
& \mathtt{term}(sl(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)))
\end{aligned}
$$

The second property is derived from the first. (1) If $sl_n(\ldots sl_1(t)\ldots)$ is defined, then $\mathtt{term}(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots))$ is defined because of the strong equality. Because the interpretation of `term` is strict, $sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)$ is defined as well. (2) If $sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots)$ is defined, then $\mathtt{term}(sl_n(\ldots sl_1(\mathtt{pos}(t))\ldots))$ is defined, because `term` is specified as a total function. Because of strong equality, the second property yields that $sl_n(\ldots sl_1(t)\ldots)$ is defined as well. QED

The second validation property says that the elements of the position sorts represent exactly the valid selection paths for the terms of sort R_1:

Lemma 2. Let $\mathrm{MetaPos}(\mathrm{R}_i)$ be the set of valid selection paths from a term t of sort R_1 to a subterm of sort R_i and let $\mathrm{R}_i^{\pi}(\mathcal{A})$ denote the carrier set of sort R_i^{π} in $\mathcal{A}$. Then the following mappings ρ_i, $i \in \{1, \ldots, q\}$, are bijective:

$$
\begin{aligned}
& \rho_i \;:\; \mathrm{MetaPos}(\mathrm{R}_i) \;\rightarrow\; \mathrm{R}_i^{\pi}(\mathcal{A}) \\
& \rho_i(\;(t, [sl_1, \ldots, sl_n])\;) \;=_{def}\; sl_n(\ldots(sl_1(\mathtt{pos}(t)))\ldots)
\end{aligned}
$$

Proof sketch of lemma 2: We have to show that the mappings ρ_i are well-defined, injective, and surjective:
1. Well-defined: The second property of lemma 1 guarantees well-definedness.
2. Injective: It is easy to show that the standard algebra of term positions as introduced informally in Sect. 2 is a model of the specification. In that algebra, different selection paths yield different positions. Since $\mathcal{A}$ is an initial algebra, this property holds for $\mathcal{A}$ as well.
3. Surjective: According to the CASL semantics, the position sorts are generated by the constructors. That is, each element p of a position sort $\mathrm{R}_i^{\pi}(\mathcal{A})$ has a representation of the form $p = sl_k(...(sl_1(\mathtt{pos}(t)))...)$. Consequently, $sl_k(...(sl_1(\mathtt{pos}(t)))...)$ is defined. According to lemma 1, this implies that $sl_k(...sl_1(t)...)$ is defined as well. Thus, we have a preimage for each element of a position sort. QED

4.3 Extending the Tepos-Algebra Kernel

In Sect. 2, we worked with a tepos-algebra that contained more sorts and functions than the tepos-algebra kernel described above. For example, we used a function `root` and subsorts `GotoPos` and `LablPos`. Such extensions can easily be declared on top of the kernel. In CASL, their specification is straightforward. We show here only how the function `root` and some subsorts can be specified. Other examples would be a supersort for all positions and functions operating on such supersorts (for instance, a function `parent` that yields for each position the parent position). Which of these extensions are included in tepos-algebras is mainly a language design issue and beyond the scope of this paper.

We illustrate the specification of additional functions and subsorts based on the example specification SIMPLS. The function `root` can be recursively defined:

```
vars t : Prog; pp : ProgPos; sp : StmtPos; ep : ExprPos;
· ¬def root(pos(t))
· root(stm(pp)) = pp when pos(term(pp)) = pp else root(pp)
· root(fst(sp)) = root(sp)
· root(scd(sp)) = root(sp)          · root(fst(ep)) = root(ep)
· root(bod(sp)) = root(sp)          · root(scd(ep)) = root(ep)
· root(rhs(sp)) = root(sp)          · root(lhs(sp)) = root(sp)
· root(cnd(sp)) = root(sp)          · root(idt(ep)) = root(ep)
```

In a handwritten specification, the case for constructor `stm` can be simplified into `root(stm(pp))=pp`, because in the abstract syntax of SIMPL a term of sort `Prog` never occurs as a subterm. However, in general, terms of the root sort can occur as subterms. Thus, a case distinction can be necessary. Finally, we show how subsorts of sorts with multiple constructors can be specified. CASL allows to introduce new subsorts in a convenient way by set comprehension:

```
sort Assg = { t : Stmt. ∃ id : Idt, e : Expr. t = assg(id, e) }
sort AssgPos = { p : StmtPos. term(p) ∈ Assg }
...
```

All such specifications extending the tepos-algebra kernel can be generated automatically without needing any further declaration support from the user.

5 Specifying Tepos-Algebras in Higher-Order Logic

When we first looked at different specification frameworks to specify the semantics of tepos-algebras, CASL seemed not only appealing because of its support for partial functions and constructors. Appealing was as well the tool HOL-CASL [11] that allows to generate a higher-order theory from a CASL specification. Unfortunately, the current version of HOL-CASL does not support partial constructors in free specifications. Furthermore, HOL-CASL uses a special bottom element to encode partiality into HOL which only supports total functions. In our experiments, it turned out that for our verification goals it is more suitable and elegant to use a different encoding. That is why we developed our own embedding into the Isabelle/HOL framework.

The basic idea of our embedding is as follows. Partiality of a function f is handled by a definedness predicate def_f that yields true for all values on which f is defined. For values x with $\neg\, def\_f(x)$, we specify that $f(x) = arbitrary$ where *arbitrary* is some arbitrary element of the range of f. (Isabelle/HOL guarantees that sorts are nonempty and uses the Hilbert operator to formalize *arbitrary*.)

A typical application of this technique is the specification of the selectors for datatypes. The standard datatype construct of Isabelle/HOL does not support selectors. Thus, they have to be specified separately. As in Sect. 4, we use the specification SIMPLS to demonstrate the embedding. For example, the selectors *stm* and *lhs* are specified as follows:

$$
\begin{array}{l}
stm\ (x{::}Prog) \equiv case\ x\ of\ prgm\ y \Rightarrow y \\
def\_stm\ x \quad\ \equiv case\ x\ of\ prgm\ y \Rightarrow True
\end{array}
$$

$$
\begin{array}{l}
lhs\ (x{::}Stmt) \equiv case\ x\ of\ assg\ (y,\ z) \Rightarrow y \\
\qquad\qquad\qquad\qquad\quad \mid sequ\ (y,\ z) \Rightarrow arbitrary \\
\qquad\qquad\qquad\qquad\quad \mid loop\ (y,\ z) \Rightarrow arbitrary \\
def\_lhs\ x \equiv case\ x\ of\ assg\ (y,\ z) \Rightarrow True \\
\qquad\qquad\qquad\qquad\quad \mid sequ\ (y,\ z) \Rightarrow False \\
\qquad\qquad\qquad\qquad\quad \mid loop\ (y,\ z) \Rightarrow False
\end{array}
$$

Starting from the datatype and selector specification, we specify the position sorts. As Isabelle/HOL does not support partial constructors, we have to do this in several steps:

1. In the first step, we freely-generate sorts that contain more elements than we have positions. We call the sorts *ProgPosU*, *StmtPosU*, *ExprPosU*, and *IdtPosU* where "U" stands for unrestricted.
2. Then, we define functions corresponding to *term* on these sorts.
3. Using these functions, we define subsets of the unrestricted position sorts.
4. By lifting the subsets, we define the new sorts *ProgPos*, *StmtPos*, *ExprPos*, and *IdtPos*.

The datatype specification for the unrestricted sorts looks as follows – we append the character "U" to the end of the constructor names because Isabelle/HOL does not allow to overload the names of the selector functions by constructor names (for brevity, we leave out some productions):

datatype *ProgPosU* = *posU Prog*
and *StmtPosU* = *stmU ProgPosU*
| *fstU StmtPosU*
| *scdU StmtPosU*
| *bodU StmtPosU*
and *ExprPosU* = *rhsU StmtPosU*
| ...

The *term*-function is defined via primitive recursion. Since Isabelle/HOL does not allow overloading of primitive recursive functions, we specify one *term*-function for each sort. For brevity, we only show parts of the specifications and simplify the original Isabelle/HOL source a bit:

primrec
termP (*posU p*) = *p*
termS (*stmU p*) = *stm*(*termP p*)
termS (*fstU p*) = *fst*(*termS p*)
...

Using the *term*-functions, we inductively define the sets of all valid positions. Starting from a valid position, if the application of a selector on the term side is defined, then the application on the position side yields another valid position. These sets are denoted with a postfix "S" (for "set"). Again, we display only a small part of the specification.

ProgPosS :: *ProgPosU set*
inductive *ProgPosS*
(*posU x*) ∈ *ProgPosS*

StmtPosS :: *StmtPosU set*
inductive *StmtPosS*
(*x*::*ProgPosU*) ∈ *ProgPosS* ∧ (*def_stm* (*termP x*)) ⟹ (*stmU x*) ∈ *StmtPosS*
(*x*::*StmtPosU*) ∈ *StmtPosS* ∧ (*def_fst* (*termS x*)) ⟹ (*fstU x*) ∈ *StmtPosS*

Isabelle enables to specify types/sorts[4] for such sets provided the sets can be proven to be non-empty. This can always be achieved by specifying a witness, that is an element of the set. Based on this, we can declare the sorts *ProgPos*, *StmtPos*, etc.:

typedef *ProgPos* = *ProgPosS*
typedef *StmtPos* = *StmtPosS*

For these types, Isabelle/HOL automatically provides us with representation and abstraction functions. For example, the representation function *Rep_ProgPos* takes an argument of type *ProgPos* and yields the corresponding element of

[4] In Isabelle, sorts are called types.

the underlying set *ProgPosS*, that is, it has range type *ProgPosU*. The abstraction function *Abs_ProgPos* has domain type *ProgPosU*. It maps elements from *ProgPosS* to their abstraction in *ProgPos*. Elements not contained in *ProgPosS* are mapped to *arbitrary*. (Additionally, Isabelle provides a number of lemmas regarding injectivity, inversion, and so on.)

Finally, we have to lift the functions on positions to the new restricted sorts. We demonstrate this here for *fstU*. The corresponding partial function from *StmtPos* to *StmtPos* is denoted by *fstP* ("P" for "partial"). *fstP* and the corresponding definedness predicate *def_fstP* are defined as follows:

$$fstP\ y \equiv Abs\_StmtPos\ (fstU\ (Rep\_StmtPos\ y))$$
$$def\_fstP\ y \equiv (fstU\ (Rep\_StmtPos\ y)) \in StmtPosS$$

Discussion. It is interesting to compare the CASL and the Isabelle/HOL specifications. The CASL specification is much shorter and, what is more important, the underlying idea of the specification technique is directly visible. This is possible because CASL supports partial constructors in a free specification where the partiality depends on an inductively specified total function. Of course, this elegance comes at the price that consistency checking of the specification is more complex. Whereas the Isabelle/HOL theory provides by construction a conservative extension of the datatype specifying the term algebra, extensions in CASL can lead to inconsistent specifications. Considering our work as a specification case study, we learned two lessons:

1. It is helpful to start with a loose specification. We first tried to develop the formalization of tepos-algebras directly in HOL – and almost gave up. Then, we learned about CASL and that its nice, well-integrated features allow for a very concise specification. This was the step when we identified the kernel of tepos-algebras. Finally, we could construct a HOL specification, focussing on design issues simplifying verification.
2. Formalizing partial functions by adding a bottom element to the range and domain types (e.g. by using the type constructor *option*) is not always a good choice. Using the Hilbert-operator and a definedness predicate can lead to more practical specifications, that is, to specifications that simplify the formal verification using interactive tactical provers.

6 Related Work

To our knowledge, this is the first work on formal specification of sorted term positions at the object level. We developed tepos-algebras as a foundation for language specification and implementation tools. Having a rich tree representation enables to use language specification techniques that do not work for free constructor terms. That is why most language specifications with abstract state machines are based on such rich tree represenations (as one example, see [12]).

Depending on the application area, other tree representations and formalization techniques are used. Higher-order abstract syntax (see [13]) is particularly well suited for matching, substitution in terms, and unification. It allows to ab-

stract over parameterized subterm positions. This is very helpful to express name bindings and consistent renamings. So far, we have not looked at how practical it is to specify substitution in a tepos-algebra framework.

Special logics become more and more popular to describe certain kinds and properties of trees or to discover the shape of trees. The logic underlying Mona [14] can for example be used to describe pointer structures as part of a decidable program logic. Similarly, shape analysis uses a logic as a basis for automated analyses for programs with pointers (see e.g. [15]).

7 Conclusions

We demonstrated how tepos-algebras can be used and formally specified. As application area, we looked at language specifications and showed how a continuation semantics can be avoided if the abstract syntax trees of the language are represented by a tepos-algebra. Similarly, complex environments can be avoided by using the position of the declaration to access the declaration information of a program elements.

The main part of the paper explained shallow embeddings of tepos-algebras into CASL and into Isabelle/HOL. Our conclusion is that such frameworks should be used in combination. The powerful CASL language allows to exploit and compare different specification techniques, which is very helpful in the design phase of the specification. On the other hand, Isabelle/HOL provides more automated checks for the specification.[5] Furthermore, it enables to refine the specification towards effective verification applications. Future work in that direction is the development of proof principles and proof strategies for tepos-algebras.

References

1. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)
2. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Springer-Verlag (1993)
3. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: The Common Algebraic Specification Language. Theoretical Computer Science **286** (2002) 153–196
4. Bidoit, M., Mosses, P.D.: CASL User Manual. LNCS 2900. Springer-Verlag (2004)
5. CoFI (The Common Framework Initiative): CASL Reference Manual. LNCS 2960. Springer-Verlag (2004)
6. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283. Springer-Verlag (2002)
7. Poetzsch-Heffter, A.: Prototyping realistic programming languages based on formal specifications. Acta Informatica **34** (1997) 737–772

[5] There are as well tools for consistency checking of CASL specifications (see www.informatik.uni-bremen.de/cofi/ccc/). However, we had problems to apply them to specifications with partial constructors.

8. Bauer, B., Höllerer, R.: Übersetzung objektorientierter Programmiersprachen. Springer-Verlag (1998)
9. Slonneger, K.: Executing continuation semantics: A comparison. Software – Practice and Experience **23** (1993) 1379–1397
10. Roggenbach, M., Schröder, L.: Towards trustworthy specifications I: Consistency checks. In Cerioli, M., Reggio, G., eds.: Recent Trends in Algebraic Specification Techniques, 15th International Workshop, WADT 2001. LNCS 2267, Springer-Verlag (2001)
11. Mossakowski, T.: Introduction into HOL-CASL (Version 0.82). Technical report, University of Bremen (2002)
12. Börger, E., Schulte, W.: Programmer Friendly Modular Definition of the Semantics of Java. In Alves-Foss, J., ed.: Formal Syntax and Semantics of Java. LNCS 1523. Springer-Verlag (1998)
13. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In Wise, D.S., ed.: SIGPLAN '88 Conference on Progamming Language Design and Implementation. SIGPLAN Notices 23(7), ACM Press (1988) 199–208
14. Klarlund, N.: Mona & Fido: The logic-automaton connection in practice. In: Computer Science Logic, CSL '97. LNCS 1414, Springer-Verlag (1998) 311–326
15. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1996) 16–31

A Declaration of Function `target`

The declaration of function `target` shows how functions can be used to represent references from one tree node to another in a declarative way:

```
datatype  LablPosNil = lbpos( lbp: LablPos )   |  nil  end

labl_lkup: Idt x StmtPos -> LablPosNil
  labl_lkup(id,sm) = case sm of
    sequ<sm1,sm2> =>  if labl_lkup(id,sm1)!=nil then  labl_lkup(id,sm1)
                                                else  labl_lkup(id,sm2)
  | loop<e,body>  =>  labl_lkup(id,body)
  | labl<lip,sm0> =>  if  id = term(lip)  then  sm
                                          else  labl_lkup(id,sm0)
  | _             =>  nil

target: GotoPos -> LablPos
  target(gp) = lbp(labl_lkup(term(tid(gp)), stm(root(gp))))
```