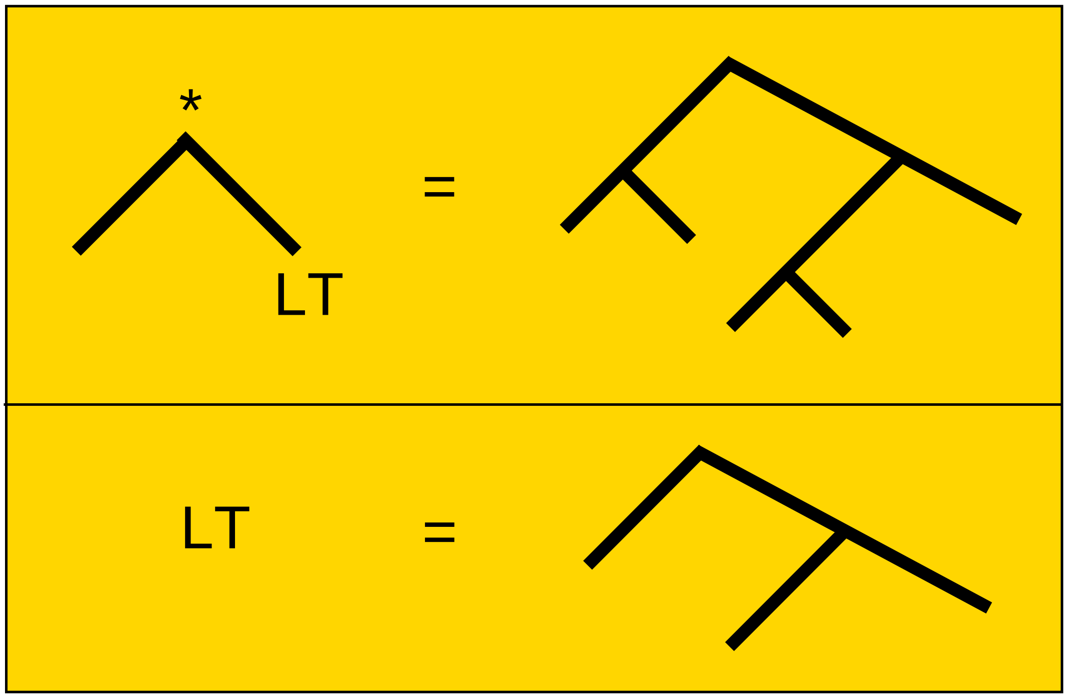
University of Amsterdam

Programming Research Group



Solving Type Equations in Multi-Level Specifications (preliminary version)

E. Visser

Report P9606

May 1996

University of Amsterdam

Department of Computer Science

Programming Research Group

Solving type equations in multi-level specifications (preliminary version)

E. Visser

Report P9606 May 1996

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@fwi.uva.nl

The research was conducted with support from the Dutch Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

Universiteit van Amsterdam, 1996

Solving Type Equations in Multi-Level Specifications
(preliminary version)

Eelco Visser
Programming Research Group, Department of Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, tel: +31 20 525 7590, fax: +31 20 525 7490, email: visser@fwi.uva.nl

May 7, 1996

Abstract

In first-order algebraic specification functions have types of the form $s_1 \times \cdots \times s_n \to s_0$, where the s_i are type constants. Such types exclude higher-order and polymorphic functions. In multi-level algebraic specification the structure of types used in function declarations is specified as an algebraic data type. If only free constructors are used in the types used in function declarations, type assignment is an extension of the Hindley/Milner algorithm to multiple levels of types. By means of equations over types, sophisticated type systems can be modeled in a simple and uniform language. The type assignment for arbitrary multi-level specifications requires $\mathcal{E}$-unification. Although this is undecidable in general, it is decidable for restricted sets of equations.

In an earlier paper, the modular applicative multi-level equational specification formalism MLS is defined. The typechecker supports only free type constructors.

In this paper we introduce multi-level specification by means of a series of MLS examples and discuss the extension with an $\mathcal{E}$-unification procedure instead of syntactic unification of the type assignment function for MLS such that it supports type definitions, defined type operators and recursive types.

1 Introduction

Types denote sets of values and type declaration divide values in categories. Typechecking helps debugging programs and specifications by finding violations of this categorization, but restricts the flexibility of a program. Therefore, many typing constructs are designed that are flexible and admit a decidable type system. In this paper we look at a uniform and simple extension of algebraic specification formalisms providing a more flexible type system and capable of expressing many type constructs.

In first-order algebraic specification formalisms the types of functions are terms of the form $s_1 \times \cdots \times s_n \to s_0$, denoting first-order monomorphic functions with domain $s_1 \times \cdots \times s_n$ and codomain s_0, where the s_i are type constants.

Typechecking consists in verifying that the types of the actual arguments of a function correspond to the formal arguments. In functional programming languages types are terms over the signature $\tau ::= \alpha \mid f(\tau_1, \ldots, \tau_n) \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$ denoting higher-order polymorphic types. This type structure is a first-order *one-sorted* data type. The type constructors f have 'kind' $\mathtt{type}^n \to \mathtt{type}$, where `type` is the implicit type of types. The Hindley/Milner type assignment algorithm (Milner, 1978) associates a principal type with each well-formed expression.

In *two-level algebraic specification* (Poigné, 1986; Möller, 1987; Meinke, 1992) the structure of types is a user-defined abstract data type defined by means of a first-order many-sorted algebraic specification. Type constants and constructors are declared as functions over the type of types. *Multi-level specification formalisms* (Hearn and Meinke, 1994; Hearn, 1995; Visser, 1996) generalize two-level specifications by allowing the specification of arbitrary many levels of types. This provides the possibility to model many type constructions in a single, simple framework.

If the algebra of types is freely generated from a signature, the type assignment can use syntactic unification and reduces to an extension of the Hindley/Milner algorithm to multiple levels of types. If equations over types are present the type assignment problem requires $\mathcal{E}$-unification.

In Visser (1996) the modular applicative multi-level equational specification formalism MLS is defined. The typechecker in that definition only considers free type constructors. A similar formalism is ATLAS (Hearn and Meinke, 1994; Hearn, 1995), which uses algebraic instead of applicative term structure.

In this paper we extend the typechecker for MLS with an $\mathcal{E}$-unification procedure. In the formalism supported by the typechecker one can model type definitions, type operations (such as mapping a type constructor over a list of types) and recursive types.

In the next two sections we give an informal introduction to MLS by means of a number of examples. In Section 2 we introduce multi-level specifications with free types. In Section 3 we give examples of multi-level specifications with equations over types. In Sections 4 until 8 we discuss the type system and unification procedure.

2 Multi-Level Specifications

In this section we explain multi-level specification by means of several examples. As our language we use the formalism MLS of Visser (1996).

2.1 One-Level Specification

A one-level specification consists of a many-sorted signature, introducing sorts (a synonym for types) and functions over those sorts, and a list of equations over the terms generated by the signature. The functions defined in a one-level specification are monomorphic. The next module shows how the well-known successor natural numbers are specified by one-level MLS specification.

```
module nat
level 0
```

```
  signature
    sorts nat;
    functions
      0   : nat;
      s   : nat -> nat;
      (+) : nat # nat -> nat;
    variables
      I, J : nat;
  equations
    0 + I    == I;
    s(I) + J == s(I + J);
```

2.2 Type Constructors

In the same way that natural numbers are specified, we can specify a signature of `type`s introducing two binary (infix) type constructors `(#)` for product types and `(->)` for function types. The signature for `type` is defined at level 1, which entails that terms over this signature can be used at level 0 as sorts.

```
module type
level 1
  signature
    sorts type;
    functions
      (#), (->) : type # type -> type;
    variables
       A, B, C : type;
```

In module `nat_typed`, the natural numbers as specified in module `nat` are incorporated in the world of `type`s by declaring `nat` as a `type` constant. The sort expressions used in module `nat` are now `type` terms. The declaration of the `type` variable `A` serves to allow any term of type `type` as a sort at level 0. Such a declaration excludes non-`type` terms like `(#)` as level 0 sorts.

```
module nat_typed
imports type, nat;
level 1
  signature
    sorts A;
    functions
      nat : type;
```

Observe that in module `nat` and module `type` the sorts used at level 0 and level 1, respectively, are not defined by means of a signature. The sorts used at the highest level of a multi-level specification are determined by an implicit signature consisting of the declaration

```
        (#), (->) : top # top -> top
```

and for each `s` the declaration `s :  top`.

2.3 Polymorphic Types

Polymorphic types can be modeled by means of type terms containing variables. We can enrich the signature of types, which consists of the basic type `nat` and the constructors `(#)` and `(->)`, by adding new type constructors. A list is a structure built by the empty list `[]` and the cons function `(::)` that adds an element to a list. A finite list of n elements is abbreviated as $[t_1, \ldots, t_n]$, which is equivalent to $t_1 :: \ldots :: t_n :: []$. The type of lists of `A`s is `list(A)`.

Many generic functions have been defined on lists. Here we give two common list functions. The function `(*)` (map) applies a function `G` to each element of a list. The function `zip` takes a pair of lists into a list of the pairs of the heads of the lists.[1]

```
module list
imports type;
level 1
  signature
    functions
      list : type -> type;
level 0
  signature
    sorts A; list(A);
    functions
      []   : list(A);
      (::) : A # list(A) -> list(A);
      (*)  : (A -> B) # list(A) -> list(B);
      zip  : list(A) # list(B) -> list(A # B);
    variables
      X : A; L : list(A); G : A -> B;
  equations
    G * []               == [];
    G * (X :: L)         == G(X) :: (G * L);
    zip([], L)           == [];
    zip(L, [])           == [];
    zip(X :: L, X' :: L') == (X, X') :: zip(L, L');
```

Instances of polymorphic functions have a type that is an instantiation of its declared type. For example, the `(::)` function for natural lists has type `nat # list(nat) -> list(nat)`. In MLS functions can also be overloaded, i.e., have more than one different type. For instance, the `(+)` operator could also be defined as concatenation operator on lists.

2.4 More Levels

List is an homogeneous data type that is parameterized by one sort. All elements of a list are members of the same sort. A tuple on the other hand is a heterogeneous structure with various types of elements. The type of tuples consists of a list of types. To specify this type we would like to reuse the specification of lists above. However, the type expressions we have used so far are

[1] Note that a variable declaration like `L : list(A)` declares all variables with 'base' `L` as `list(A)` variables, e.g., `L1`, `L2` and `L'` are also declared by this declaration.

described by a signature at the highest level of specifications. Only type constructors over the signature {`type`, `(#)`, `(->)`} can be constructed, which does not provide lists of types. We can provide more structure in the sort space of types just as we provided more structure in the sort space of values, by building yet another level. Module `kind` introduces the sort `kind` at level 2 and declares `type` as a `kind` constant.

```
module kind
imports type;
level 2
  signature
    sorts kind;
    functions
      type       : kind;
      (#), (->) : kind # kind -> kind;
    variables
      K : kind;
level 1
  signature
    sorts K : kind;
```

From here on we can proceed by declaring a constructor `list : kind -> kind` to level 2 and defining list constructors at level 1. This approach does not reuse the lists defined in module `list`. Module `type-type` also introduces the constant `type` at level 2, but uses `type` itself as its type! The `type`s defined in module `type` are used as 'kinds', by *lifting* all specifications one level. Now we can reuse all `type` constructors defined so far for level 1 at level 2, by simply lifting their specification.

```
module type-type
lift(imports type);
imports type;
level 2
  signature
    functions
      type : type;
```

For instance, to construct lists of types we import the definition of level 0 lists in module `list` and lift it to the level 1. The type of tuples is now defined by means of the type constructor `prod` that constructs a generalized product type from a list of types. A tuple is constructed by means of the functions `<>` (empty tuple) and `(^)`, which adds an element to a tuple. (Note that `<X1, ..., Xn>` is an abbreviation for `X1 ^ ... ^ Xn ^ <>`.) For instance, the tuple `<0, [0], t>` has type `prod([nat, list(nat), bool])`. The first element of a tuple is given by `exl` and the rest by `exr`.

```
module generalized-product
imports type-type;
lift(imports list);
level 1
  signature
```

```
    functions
      prod : list(type) -> type;
    variables
      LT : list(type);
level 0
  signature
    sorts prod(LT);
    functions
      <>  : prod([]);
      (^) : A # prod(LT) -> prod(A :: LT);
      exl : prod(A :: LT) -> A;
      exr : prod(A :: LT) -> prod(LT);
    variables
      P : prod(LT);
  equations
    exl(X ^ P) == X;
    exr(X ^ P) == P;
```

In the Hindley/Milner type systems used in functional programming languages it is not possible to construct the type of generalized products, because only one, implicit, sort (`type`) can be used at the level 1.

3 Specifications with Type Equations

In the previous section we saw several examples of multi-level specifications with free type constructors. In this section we look at several applications of equations over types.

3.1 Type Definitions

Abbreviations are a first application of type equations. A type can be defined as an abbrevation of a complicated type. For instance, the following module defines the type operator `parser` that takes two types and constructs the type of a combinator parser, i.e., a function that takes a list and returns a pair of some result (a tree, for instance) and the rest of the list. The functions `item` and `seq`[2] are examples of combinators that combine parsers into new parsers. The first constructs a parser that matches a single element of a list and the second parses a prefix of the list with parser `P1` and a prefix of the rest with `P2`. This module make use of several imported functions on Booleans, Cartesian products and lists not shown here: conditional, equality, identity (`i`), function composition (`.`), left and right projections (`exl` and `exr`) and `split` that takes two functions and produces the pair of the two applications to the same argument.

```
module parser_combinator
imports type, list, product;
level 1
  signature
    functions
```

[2]Note that $t_1.t_2.t3$ is an abbreviation of for $t_2(t_1, t_3)$.

```
      parser : type # type -> type;
  equations
    parser(A, B) == list(A) -> list(B # list(A));
level 0
  signature
    functions
      item : A -> parser(A, A);
      seq  : parser(A, B) # parser(A, C) -> parser(A, B # C);
    variables
      P : parser(A, B); L : list(A); X, Y : A;
  equations
    item(X)([])     == [];
    item(X)(Y :: L) == if(eq(X, Y), (X, L) :: [], []);
    (P1 .seq. P2)(L)
        == concat (((*) . ((#) . (pair .split. k(i)) # i))
                  * ((i # P2) * P1(L)));
```

3.2 Recursive Types

The type definitions of the previous example can be eliminated by simply applying the definition as a rewrite rule. The following example uses recursive type equations to define the type of binary trees in terms of disjoint sums and Cartesian products. All functions of the tree abstract data type are implemented in terms of the primitives for sums and products. The characteristic functions of sums are the left and right injection functions `inl` and `inr` and the selection function `case`. The function `[]` is a constant of type `[]`, denoting the empty type. It is clear that the definition of `tree` can not be eliminated, since this leads to an infinite term.

```
module rec_tree
imports sum, product, bool, nat;
level 1
  signature
    functions
      tree : type -> type;
  equations
    tree(A) == [] + (tree(A) # A # tree(A));
level 0
  signature
    functions
      empty       : tree(A);
      node        : tree(A) # A # tree(A) -> tree(A);
      left, right : tree(A) -> tree(A);
      val         : A -> tree(A) -> A;
  equations
    left   == i .case. exl;            empty == inl([]);
    right  == i .case. (exr . exr);    node  == inr;
    val(A) == k(A) .case. (exl . exr);
```

3.3 Type Functions

The function `zip` as defined in module `list` in Section 2 takes a pair of lists into a list of pairs by pairing the heads of both lists until one of the lists is empty. In Hindley/Milner type systems it is not possible to construct a definition of `zip` for arbitrary tuples. Therefore, funcional programs define `zip3`, `zip4` etc. In Section 2 we defined generalized products. The following module defines a generalization of `zip` for tuples of lists. For instance, if the argument of `zip` has type `prod([list(nat), list(bool), list(list(bool))])`, its result has type `list(prod([nat, bool, list(bool)]))`.

```
module generalized-zip
imports generalized-product, list;
level 0
  signature
    functions
      zip : prod(list * LT) -> list(prod(LT));
    variables
      L, M, N : list(A);
  equations
    zip(<>)                == []; -- empty tuple
    zip(<[]>)              == []; -- singleton tuple
    zip(<X :: L>)          == <X> :: zip(<L>);
    zip(<[], L>)           == []; -- pair
    zip(<L, []>)           == [];
    zip(<X :: L, Y :: M>)  == <X, Y> :: zip(<L, M>);
    zip(L ^ (M ^ (N ^ P))) == zip(L ^ zip(M ^ (N ^ P)));-- triple+
```

The declaration of `zip` relates the `list` types in the domain to the types in the codomain. This is achieved by declaring the domain as `prod(list * LT)`. The type constructor `list` is mapped using type function `(*)` (defined in module `list`) over the list of types `LT`. This enforces the requirement that all elements of the argument tuple of `zip` are lists. For the example above, we can see that, according to the equations for `(*)`, `prod([list(nat), list(bool), list(list(bool))])` is equal to `prod(list * [nat, bool, list(bool))])`.

3.4 Type Constraints

Another example of a specification that uses equations over types, is the following module that models the restriction of the polymorphism of the equality function by means of a mechanism like the type classes of Haskell (Wadler and Blott, 1989). The module imports module `bool` that defines the standard operations on the Boolean values `t` (true) and `f` (false). At level 1 a unary boolean function (a predicate) `eq` on types is defined such that the type `nat` is in the `eq` class and such that a list type is in the `eq` class if its content type is in the class. The operator `(=>)` constrains a type by some boolean condition. At level 0 the equality function `eq` is now declared with type `eq(A) => (A # A -> bool)`, which expresses that the function only applies to types in the `eq` class. The function `(!)` is used to apply a function with a constrained type to an argument. It requires that the condition is equal to `t`. This ensures that `eq` cannot be applied to function types or other types not in the `eq` class.

```
level 1
  signature
    functions
      eq   : type -> bool;
      (=>) : bool # type -> type;
   equations
     eq(nat) == t; eq(list(A)) == eq(A);
level 0
  signature
    functions
      (!) : (t => A -> B) # A -> B;
      eq  : eq(A) => (A # A -> bool);
equations
  eq!([], [])           == t;
  eq!(X :: L, [])       == f;
  eq!([], X :: L)       == f;
  eq!(X :: L, X' :: L') == eq!(X, X') /\ eq!(L, L');
```

Other type constructs that can be defined in a similar way are record types. The type assignment function defined in the next sections supports simple type definitions, recursive type definitions and type functions, but does not support type class constraints. This is due to the fact that the rhs of the equation for lists is more general than the lhs of that equation, leading to an infinite chain of rewrites. Such chains should be detected and the solution of the constraint deferred until more information about the argument of `eq` is known.

4 Terms and Types

In the next sections we discuss the syntax and semantics of multi-level specifications and in particular the extension of the syntactic unification function to $\mathcal{E}$-unification.

The syntax of terms is defined by the following SDF definition. (see Bergstra *et al.*, 1989; Klint, 1995, for an introduction to ASF+SDF). The productions should be read as function declarations, taking arguments on the lhs and producing a result of the sort on the rhs. Terms are built from variables, functions, application, pairing and annotation (:). Binary infix operators are abbreviations of binary prefix functions and are translated by $t_1 \oplus t_2 = (\oplus)(t_1, t_2)$. Types are represented in the same syntax. The annotation operator is used to attach type terms to the subterms of a term. A term is fully annotated if each subterm has an annotation. The term(t) of a term t is the term without its annotation, the type(t) of a term t is its annotation. A term without explicit annotation has the implicit type 'top'.

```
exports
  sorts Fun Var Term
  lexical syntax
    [a-z0-9][A-Za-z0-9_]*  → Fun
    [A-Z][A-Za-z_]*[0-9']*  → Var
  context-free syntax
    Var                → Term
    Fun                → Term
    Term Term          → Term {left}
    Term "," Term      → Term {right}
    Term ":" Term      → Term {right}
    "(" Term ")"       → Term {bracket}
  priorities
    Term Term → Term > ":" > ","
  variables
    [xy][0-9']* → Var
    "f"[0-9']* → Fun
    "t"[0-9']* → Term
    "τ"[0-9']* → Term
```

5 Multi-Level Equational Logic

The rules for multi-level equational logic are defined in Table 1. The rules define the predicate $\Gamma \vdash t_1 \equiv t_2$, indicating that equation $t_1 \equiv t_2$ is derivable from the multi-level specification Γ. The rules are defined in ASF+SDF, by means of conditional equations. A predicate in ASF+SDF is just a Boolean function. The value $\top$ denotes truth. An equation is an axiom for a multi-level specification Γ if it is an equation at level 0. $\mathrm{E}(\pi_0(\Gamma))$ is the list of equations of level 0 of Γ. Rule [ann] states that two annotated terms are equal if their terms are equal and if their annotations are equal over the next level. The function 'up'(Γ) drops level 0 from Γ and decreases all higher levels by one. The standard rules for equational logics are the reflexivity, symmetry, transitivity and substitution rules.

[ax]	$\dfrac{\mathrm{E}(\pi_0(\Gamma)) = \varphi_1^*;\ t_1 \equiv t_2;\ \varphi_2^*}{\Gamma \vdash t_1 \equiv t_2\ =\ \top}$	$\Gamma \vdash t \equiv t\ =\ \top$	[refl]
[ann]	$\dfrac{\Gamma \vdash t_1 \equiv t_2 = \top,\ \mathrm{up}(\Gamma) \vdash \tau_1 \equiv \tau_2 = \top}{\Gamma \vdash t_1 : \tau_1 \equiv t_2 : \tau_2\ =\ \top}$	$\dfrac{\Gamma \vdash t_2 \equiv t_1 = \top}{\Gamma \vdash t_1 \equiv t_2\ =\ \top}$	[sym]
[trans]	$\dfrac{\Gamma \vdash t_1 \equiv t_2 = \top,\ \Gamma \vdash t_2 \equiv t_3 = \top}{\Gamma \vdash t_1 \equiv t_3\ =\ \top}$	$\dfrac{\Gamma \vdash t_1 \equiv t_2 = \top}{\Gamma \vdash \sigma(t_1) \equiv \sigma(t_2)\ =\ \top}$	[sub]

Table 1: Multi-level equational logic.

The unification problem for multi-level specifications can now be defined as follows. Given an equation $t_1 \equiv t_2$ and a specification Γ find a most general unifier σ such that $\Gamma \vdash t_1 \equiv t_2$. The well-formedness and type assignment rules in the next sections are parameterized with unification functions.

6 Well-formedness

Typechecking of multi-level specifications is defined in two parts. The well-formedness predicates of this section define which fully annotated terms are well-formed. The type assignment functions of the next section map plain terms as written by a programmer to fully annotated terms. The full set of rules for well-formedness of specifications and signatures with free type constructors are defined in Visser (1996). Here we only discuss the rules for terms and sorts; the part where unification plays a role.

[wf-term]	$\Gamma \models_{\mathrm{term}} t \;=\; \Gamma \models_{\mathrm{trm}} \mathrm{term}(t) : \mathrm{type}(t)$
[wf-top]	$\Gamma \models_{\mathrm{trm}} \mathrm{top} : \mathrm{top} \;=\; \top$
[wf-fun]	$\dfrac{\pi_f(\Gamma) \geq \tau = \top}{\Gamma \models_{\mathrm{trm}} f : \tau \;=\; \top}$
[wf-var]	$\dfrac{\pi_x(\Gamma) = \Phi,\;\; \Phi \geq \tau \vee \mathrm{empty}(\Phi) = \top}{\Gamma \models_{\mathrm{trm}} x : \tau \;=\; \Gamma \models_{\mathrm{sort}} \tau}$
[wf-pr]	$\dfrac{\mathrm{term}(\tau) \doteq_{\mathrm{up}(\Gamma)} \mathrm{type}(t_1) \times \mathrm{type}(t_2) = \top}{\Gamma \models_{\mathrm{trm}} t_1, t_2 : \tau \;=\; \Gamma \models_{\mathrm{term}} t_1 \wedge \Gamma \models_{\mathrm{term}} t_2}$
[wf-app]	$\dfrac{\mathrm{type}(t_1) \doteq_{\mathrm{up}(\Gamma)} \mathrm{type}(t_2) \rightarrow \tau : \mathrm{type}(\mathrm{type}(t_1)) = \top}{\Gamma \models_{\mathrm{trm}} t_1\; t_2 : \tau \;=\; \Gamma \models_{\mathrm{term}} t_1 \wedge \Gamma \models_{\mathrm{term}} t_2}$

Table 2: Well-formedness of terms.

The well-formedness rules for terms are shown in Table 2. The predicate $\models_{\mathrm{term}}$ is defined in terms of $\models_{\mathrm{trm}}$ by making the possibly implicit type annotation of a term explicit, to deal with terms over all levels in a uniform way. The function $\pi_f(\Gamma)$ yields the set of types for function f in level 0 of Γ. The predicate $\Phi \geq \tau$ tests whether term τ is an instance of one of the terms in set Φ. The predicate $t_1 \doteq_\Gamma t_2$ tests whether terms t_1 and t_2 are equal in Γ, i.e., if $\Gamma \vdash t_1 \equiv t_2$.

The central idea of multi-level specifications is that a sort is well-formed iff it is a well-formed term over the next level. The rules for sort well-formedness are defined in Table 3. The first rule states that any term over the implicit signature on top of a specification is a sort in the highest level. The second states that a sort at a lower level should be an instance of one of the sorts in the sort declarations of that level and should be a well-formed term over the next level.

[wf-sort1]	$\dfrac{\mathrm{zero}(\max(\Gamma)) = \top}{\Gamma \models_{\mathrm{sort}} t \;=\; \mathrm{up}(\Gamma) \models_{\mathrm{term}} t}$
[wf-sort2]	$\dfrac{\mathrm{zero}(\max(\Gamma)) = \bot,\;\; \{\mathrm{S}(\mathrm{Sg}(\pi_0(\Gamma)))\} \geq t = \top}{\Gamma \models_{\mathrm{sort}} t \;=\; \mathrm{up}(\Gamma) \models_{\mathrm{term}} t}$

Table 3: Well-formedness of sorts.

7 Type Assignment

The type assignment function $\mathrm{Wt}(\Gamma)[\![t]\!]$ assigns types to term t given the declarations in Γ. The function yields the set of all possible, most general, typings of t. More than one typing of a term can be the result if functions in the term are overloaded. The most interesting case are the annotation of functions and application, which are defined in Table 4.

A function is annotated with the complete set of types assigned to it in declarations. An application is assigned by assigning its subtermss. The function $\bowtie$ first renames the type variables of the sets of terms and then produces the Cartesian product of the resulting sets. The function 'new-arrow' produces a function type with new undeclared variables and annotates it as a sort. The function 'app' determines for each triple in the resulting set, whether it constitutes a valid application by (semantically) unifying the type of the function with the arrow type and the type of the argument with the domain type. If the result is succesful the codomain of the arrow is the type annotation of the application. Observe that the substitution σ is applied to the entire application, which entails that type variables in annotations in subterms can become instantiated.

[wt-fun]	$\mathrm{Wt}(\Gamma)[\![f]\!] \;=\; f : \pi_f(\Gamma)$
[wt-app]	$\mathrm{Wt}(\Gamma)[\![t_1\ t_2]\!] \;=\; \mathrm{app}(\mathrm{up}(\Gamma)) * ((\mathrm{Wt}(\Gamma)[\![t_1]\!] \bowtie \mathrm{Wt}(\Gamma)[\![t_2]\!]) \bowtie \{\mathtt{a} : \text{new-arrow}(\Gamma)\}\ )$
[app1]	$\dfrac{\mathrm{mgu}(\Gamma)(\mathrm{type}(t_1) \equiv \tau_1;\, \mathrm{type}(t_2) \equiv \tau_2) = \sigma}{\mathrm{app}(\Gamma)((t_1,\, t_2),\, \mathtt{a} : (\tau_1,\, \tau_2,\, \tau_3)) \;=\; \langle \top,\, \{\sigma(t_1\ t_2 : \tau_3)\}\rangle}$
[app2]	$\mathrm{app}(\Gamma)((t_1,\, t_2),\, \tau) \;=\; \langle \bot,\, \{t_1\ t_2 : \mathrm{nil}\}\rangle$ **otherwise**
where new-arrow$(\Gamma) = (x \to x', x, x')$ with x, x' new variables	

Table 4: Type assignment for functions and application.

8 E-Unification

Finally, Tables 5 and 6 give the rules for $\mathcal{E}$-unification that solve the equations for simple type definitions, recursive type definitions and type functions. (For a survey of unification techniques see Jouannaud and Kirchner (1991).) Table 5 gives the rules for syntactic unification that decompose a term if the function symbols on both sides are the same. The rules in Table 6 extend consider the equations of a specification Γ in unifying a list of equations.

Rule [eu-ann] unifies two annotated terms. The annotations should unify with respect to the next level of the specification. Rules [eu-red-i] apply equations as rewrite rules from left to right. If one of the terms t_i syntactically unifies with the lhs of an equation at level 0, then it can be replaced by the instantiation of the rhs of the equation. The resulting substitution is restricted to the variables in the original problem. All rules are backtracking rules; if the unification of the smaller problem does not succeed, the condition fails, and some other rule is tried for the original problem, until no more apply. This strategy works if every defined function results in a term with a free constructor as top symbol after a finite number of steps, even for open terms. However, it does not work for the type constraints of the type class specification.

[eu-refl] $$\mathrm{mgu}(\Gamma)(t \equiv t) \;=\; []$$

[eu-var] $$\frac{x \in \mathrm{vars}(t) = \bot}{\mathrm{mgu}(\Gamma)(x \equiv t) \;=\; [x := t]}$$

[eu-var'] $$\frac{x \in \mathrm{vars}(t) = \bot}{\mathrm{mgu}(\Gamma)(t \equiv x) \;=\; [x := t]}$$

[eu-bin] $$\frac{\mathrm{mgu}(\Gamma)(t_1 \equiv t_3;\, t_2 \equiv t_4) = \sigma}{\mathrm{mgu}(\Gamma)(t_1 \oplus t_2 \equiv t_3 \oplus t_4) \;=\; \sigma}$$

where $\oplus \in \{(@), (,), (\rightarrow), (\times)\}$ with $(@)$ the application operator.

[eu-es-0] $$\mathrm{mgu}(\Gamma)() \;=\; []$$

[eu-es-n] $$\frac{\mathrm{mgu}(\Gamma)(\varphi_1^+) = \sigma_1,\ \mathrm{mgu}(\Gamma)(\sigma_1 * e(\varphi_2^+)) = \sigma_2}{\mathrm{mgu}(\Gamma)(\varphi_1^+;\, \varphi_2^+) \;=\; \Downarrow(\sigma_2 \circ \sigma_1)}$$

Table 5: Rules for unification.

[eu-ann] $$\frac{\begin{array}{c}\mathrm{mgu}(\mathrm{up}(\Gamma))(t_2 \equiv t_4) = \sigma_1,\\ \mathrm{mgu}(\Gamma)(\sigma_1 * e(t_1 \equiv t_3)) = \sigma_2,\ \sigma_2 \circ \sigma_1 = \sigma_3\end{array}}{\mathrm{mgu}(\Gamma)(t_1 : t_2 \equiv t_3 : t_4) \;=\; \sigma_3}$$

[eu-red-l] $$\frac{\mathrm{red}(\Gamma)(t_1 \equiv t_2) = \sigma}{\mathrm{mgu}(\Gamma)(t_1 \equiv t_2) \;=\; \sigma}$$

[eu-red-r] $$\frac{\mathrm{red}(\Gamma)(t_2 \equiv t_1) = \sigma}{\mathrm{mgu}(\Gamma)(t_1 \equiv t_2) \;=\; \sigma}$$

[eu-red] $$\frac{\begin{array}{c}\mathrm{E}(\pi_0(\Gamma)) = \varphi_1^*;\, t_3' \equiv t_4';\, \varphi_2^*,\\ \mathrm{vars}(t_1, t_2) = \Phi,\ \mathrm{rn\ vars}(t_3', t_4')[\Phi](t_3' \equiv t_4') = t_3 \equiv t_4,\\ \mathrm{mgu}(t_1 \equiv t_3) = \sigma_1,\ \mathrm{mgu}(\Gamma)(\sigma_1(t_4 \equiv t_2)) = \sigma_2,\ \sigma_2 \circ \sigma_1 = \sigma_3\end{array}}{\mathrm{red}(\Gamma)(t_1 \equiv t_2) \;=\; \sigma_3 \mid \Phi}$$

Table 6: Rules for $\mathcal{E}$-unification.

The following example shows how the unification of the types of the formal and actual argument of the `zip` function works. (Where we have chosen a succesful path.)

```
                    prod(list * LT) == prod(list(bool) :: [])
                ==> list * LT == list(bool) :: []
[LT := B :: LT'] ==> list(B) :: (list * LT') == list(bool) :: []
                ==> list(B) == list(bool); list * LT' == []
[B := bool LT' := []] ==>
[LT := bool :: []]
```

9 Concluding Remarks

Multi-level specifications can be used to model advanced type constructs by means of a simple and uniform formalism. Many such constructs can be described in a non-trivial fragment of multi-level specifications that is much stronger than Hindley/Milner systems, yet decidable. The syntax and type system of the multi-level specification formalism MLS is specified as an executable specification in ASF+SDF and can be used to typecheck, execute or compile multi-level specifications. The extension of the unification and type assignment algorithms to cope with type definitions *and* type constraints such as arise in the specification of type classes, is an interesting problem for further research.

References

Bergstra, J. A., Heering, J., and Klint, P., editors (1989). *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley.

Hearn, B. M. (1995). *The Design and Implementation of Typed Languages for Algebraic Specification.* Ph.D. thesis, University of Wales, Swansea.

Hearn, B. M. and Meinke, K. (1994). ATLAS: A typed language for algebraic specification. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Proc. First Int. Workshop on Higher-Order Algebra, Logic and Term Rewriting - HOA '93*, volume 816 of *lecture Notes in Computer Science*, pages 146–168, Berlin. Springer-Verlag.

Jouannaud, J. P. and Kirchner, C. (1991). Solving equations in abstract algebras: A rule-based survey of unification. In J. L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in Honour of Alan Robinson*, chapter 8, pages 257–321. M.I.T. Press, Cambridge (MA).

Klint, P., editor (1995). *The ASF+SDF Meta-environment User's Guide.* Version June 4. Available as ftp://ftp.cwi.nl/pub/gipe/reports/SysManual.ps.Z.

Meinke, K. (1992). Equational specification of abstract types and combinators. In E. Boerger, G. Jaeger, H. K. Buening, and M. M. Richter, editors, *Computer Science Logic - CSL'91*, volume 626 of *Lecture Notes in Computer Science*, pages 257–271, Berlin. Springer-Verlag.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3), 348–375348–375.

Möller, B. (1987). Algebraic specification with higher-order operators. In L. Meertens, editor, *Program Specification and Transformation*, pages 367–398. Elsevier Science Publishers B.V. (North-Holland).

Poigné, A. (1986). On specifications, theories, and models with higher types. *Information and Control*, **68**, 1–46.

Visser, E. (1996). Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, AMAST Series in Computing. World Scientific Publishing Co. To appear. Preprint in TR P9604, PRG, University of Amsterdam, http://www.fwi.uva.nl/research/prog/reports/.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on Principles of Programming Languages*, Austin, Texas. ACM Press.