Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Deriving Type Checkers

Martin Zuber and Fabian Linges

Bericht-Nr. 2012 – 09
ISSN 1436-9915

Deriving Type Checkers
[Technical Report 2012-09]

Martin Zuber and Fabian Linges

Technische Universität Berlin
Fakultät IV Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Ernst-Reuter-Platz 7, D-10587 Berlin

{mzuber,linges}@cs.tu-berlin.de

Abstract. The relationship between a type system's specification and the implementation of the type checker is a recurring issue when writing compilers for programming languages and it is an ongoing question if – and if so, how – the formal description of a type system can be used to support the compiler writer when implementing the type checking phase. In this paper we propose type systems formalized by constraint-based inference rules to form an ideal abstraction to accomplish the task of automatically deriving type checking functionality from them. We develop a set of algorithms employing the constraint-based flavor of the rules to perform type checks and present the design and implementation of a Haskell library utilizing these algorithms to provide functionality for the type checking phase based on the chosen abstraction.

1 Introduction

Type systems are an essential high level abstraction of modern programming languages and type checkers are standard components of their compilers. A language's type system needs to be fitted with a formal description to allow the language designer to reason about certain formal aspects of the system, e.g., well-formedness, progress, and preservation.

Questions about the relationship between a type system's formal description and its implementation have been addressed in existing work, ranging from mutual consistency between description and implementation [20,5] to the development of specification techniques suitable for generating type checkers [6]. Yet the question remains if the standard, text-book way to formalize type systems – inference rules over type judgements – can be used to derive or generate type checkers.

In this paper we focus on this question and show that a well known extension of the standard way to formalize type systems, namely the extension with constraints, can be used to derive type checking functionality. We develop an abstract type checking framework utilizing the constraint-based flavor of the type system and present the design and implementation of a Haskell library which employs our developed ideas and algorithms for deriving type checkers.

The remainder of this report proceeds as follows. In section 2, we give a precise overview of constraint-based type systems. Sections 3 and 4 present our two algorithms for generating and solving constraints. Section 5 describes the design and implementation of the library. Section 6 and 7 employ our framework to define type checkers for the languages Mini-ML and FeatherweightJava to give the reader a better understanding of the usage of the library. Section 8 presents a framework for visualizing the internals of the derived type checkers. Sections 9 and 10 describe related work and discuss some open questions regarding the implementation.

The implementation of our library, its documentation, and all example type checkers are available from [32].

2 Constraint-based Type Systems

Type systems provide a lightweight formal method to reason about programs and therefore need to be formalized in an adequate manner. A typical way to accomplish this task is to formulate type systems by inference rules.

Following notion from proof theory a typing rule consists of a sequence of premises $P_1 \ldots P_n$ and a conclusion C. Each P_i and C is a typing judgement and a rule is written with a horizontal line separating the premises from the conclusion. Typing judgements can be modeled as a ternary relation

$$\Gamma \vdash e : T$$

between a context Γ, an expression e, and a type T [4]. The turnstile $\vdash$ denotes that the type T can be derived for the expression e under the assumptions given by the context Γ. The judgements of rules may contain variables at meta level which represent an arbitrary object of a given class. A rule instance is a rule in which all meta variables have been substituted with concrete object-level pendants.

In this setting, a deduction (or derivation) is a tree of rule instances labeled with judgements. Each node is the conclusion of a rule instance and its children are the premises of the same rule instance. Thus a typing relation $\Gamma \vdash e : T$ holds if there exists a deduction of that judgement under the given set of typing rules.

This approach to formalize type systems can be extended in various ways, one particular – the extension with constraints – will be used as the underlying formalism for our approach to derive type check functionality from a type system's specification. In a type deduction step as described above it is checked whether all typing relations formulated in the premises of the instantiated rule hold. Thus in each inference step a certain set of constraints (the requirement that a typing relation holds can be seen as a constraint) is generated and directly checked. Given a constraint-based setting, instead of checking the generated constraints directly they are collected for later consideration. To capture this idea our notation for judgements in deduction rules needs to be accommodated. A constraint typing judgement can be modeled as a ternary relation extended with a constraint set

Simple Lambda

$$\Gamma \vdash x : T \mid \{T = \Gamma(x)\} \qquad (\textsc{Var})$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \mid C}{\Gamma \vdash \lambda x.e : T \mid C \cup \{T = T_1 \rightarrow T_2\}} \qquad (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : T_1 \mid C_1 \qquad \Gamma \vdash e : T_2 \mid C_2}{\Gamma \vdash (f)\, e : T \mid C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow T\}} \qquad (\textsc{App})$$

Fig. 1. Simply, constraint-based typed lambda calculus.

$$\Gamma \vdash e : T \mid C$$

and can be read as "expression e has type T under the assumptions Γ whenever the constraints in C are satisfied" [25]. In this constraint-based abstraction type checking is separated in two phases, constraint generation and constraint solving. A traversal of an abstract syntax tree generates a set of constraints and the program is well typed if and only if these constraints have a unique solution – type checking is reduced to constraint solving.

For a better understanding of this extension let us consider the constraint-based typing rules for the simply typed lambda calculus given in Fig. 1. Especially the typing rules for λ-abstraction and application illustrate the benefit of a constraint-based approach in comparison to basic inference rules: Using constraints as an abstraction allows the designer of a type system to formulate the essential consistency conditions that the type system imposes on the language [6]. Additionally, a constraint-based approach provides a certain flexibility regarding its expressiveness: the formalism can be custom-tailored to the type system to be defined by choosing the right set of constraint domains while still being strong enough to allow us to reason about the system in terms of progress and preservation.

3 Constraint Generation

To be able to use constraint-based inference rules for automatically deriving type check functionality from a given specification of the type system we need to develop an algorithm which generates – given a program and a set of typing rules – a set of constraints such that the program is well typed if and only if these constraints have a solution.

The algorithm to be presented is a modified version of Wand's type inference algorithm [33]. Thus the remainder of this section introduces Wand's algorithm in detail and discusses the needed extensions and modifications.

Wand presented the first proof that Hindley-Milner type inference can be reduced to unification by developing and proving an algorithm which proceeds in the

Input: Term t_0

Initialization: $E = \emptyset$
$G = \{(\Gamma_0, t_0, \tau_0)\}$, where τ_0 is a type variable and Γ_0 maps the free variables of t_0 to other distinct type variables

Loop: If $G = \emptyset$ then halt and return E
Otherwise: choose and delete a subgoal from G and add to E and G new verification conditions and subgoals as specified in an action table

Fig. 2. Skeleton of Wand's type inference algorithm.

manner of a verification-condition generator. His algorithm basically mimics the construction of a term's derivation tree and emits corresponding verification conditions (equations over type terms) along the way. At every step it keeps track of a set of subgoals G (the remaining type assertions to be proven) and a set E of equations over type terms which must be satisfied for the derivation to be valid. The algorithm ensures that at every stage the most general derivation tree is generated. Figure 2 captures the algorithm's basic functionality.

This generic definition of the algorithm can be completed in different ways by using different tables of actions for processing the subgoals in the loop step. Wand presented an action table for terms of the simply typed lambda calculus, which is stated in Fig. 3. In this action table three kinds of actions are defined, corresponding to the three kinds of lambda terms that might appear in the selected subgoal.

Let s be the selected subgoal.

Case 1:	$s = (\Gamma,\, x,\, t)$	Generate the equation $t = \Gamma(x)$.
Case 2:	$s = (\Gamma,\, (f)\, e,\, t)$	Let τ_1 be a fresh type variable that appears nowhere else in $(E,\, G)$. Then generate the subgoals $(\Gamma,\, f,\, \tau_1 \to t)$ and $(\Gamma,\, e,\, \tau_1)$.
Case 3:	$s = (\Gamma,\, \lambda x.\, e,\, t)$	Let τ_1 and τ_2 be fresh type variables. Generate the equation $t = \tau_1 \to \tau_2$ and the subgoal $((\Gamma, x : \tau_1),\, e,\, \tau_2)$.

Fig. 3. Action table for the typed lambda calculus.

Wand's type inference algorithm is modeled in a top-down manner: the skeleton of the algorithm describes how the construction of the derivation tree for the

```
Input:           Term $t_0$, Set of constraint-based typing rules $R$

Initialization:  $E = \emptyset$
                 $G = \{\Gamma_0 \vdash t_0 : \tau_0\}$, where $\tau_0$ is a type variable and $\Gamma_0$
                     maps the free variables of $t_0$ to other distinct type
                     variables
Loop:        If $G = \emptyset$ then halt and return $E$

             Otherwise:  1. choose and delete a subgoal $s$ from $G$
                         2. choose the best fitting rule for $s$ from $R$
                         3. $s' \leftarrow$ instantiate the chosen rule with $s$
                         4. substitute all meta-level types in $s'$ with fresh
                            type variables, add all premises in $s'$ as new sub-
                            goals to $G$ and add all constraints in $s'$ to $E$
```

Fig. 4. Constraint generation algorithm.

term t_0 is mimicked and typing equations are collected. The so called action table defines for a specific language which subgoals and verification conditions (equations) are generated at each step of the algorithm.

This action table has a distinct resemblance with constraint-based formulated type rules: based on the conclusion of the type rule premises are generated as new subgoals and type equations (equality constraints over type terms), where all meta level types of the typing rule have been replaced with fresh type variables, are recorded.

Thus Wand's algorithm is considered to form a suitable base for the library's constraint generation phase. But in order to do so, some modifications need to be made. Since the equations arising from the use of a type rule are denotated directly as a constraint in this deduction rule and the new subgoals are represented by the rule's premises, the action table of Wand's algorithm can be omitted. The skeleton now needs to be modified such that new subgoals are generated based on a rule's premises and type equations are recorded based on the constraints of a type rule where all meta level types have been replaced with fresh type variables.

A revised version of Wand's algorithm including the described modifications is given in Fig. 4.

4 Constraint Solving

Given an algorithm generating a set of constraints for an expression we now want to describe the semantics of a suitable constraint solver. But to do so, we need to introduce the notion of auxiliary functions in deduction rules first. Type systems regularly make use of auxiliary functions to define certain functionality, such as the lookup in a context or the instantiation of a type scheme, to allow a

```
Input:           Set of constraints $C$

Initialization:           $\sigma = \emptyset$
                 $(C_1, C_2) = \text{partition } C$

Loop:  If $C_1 = C_2 = \emptyset$ then halt and return $\sigma$

       If $C_1 = \emptyset$ then apply algorithm to $C_2$ and compose arisen substitution
       with $\sigma$

       Otherwise:  1. choose and delete a constraint $c$ from $C_1$
                   2. evaluate all auxiliary functions in $c$
                   3. solve $c$ and compose arisen substitution with $\sigma$
                   4. apply $\sigma$ to all constraints in $C_1$ and $C_2$
```

Fig. 5. Constraint solving algorithm.

more concise description of a system's type rules. These auxiliary functions are not necessarily specified in an inference rule style which yields the designer of a type system additional flexibility when defining deduction rules. We consider the notion of auxiliary functions quite useful and want to extend our model such that we allow auxiliary functions not only in a rule's premises, but also in constraints. We even go one step further and allow auxiliary functions in constraints whose correct evaluation might depend on the solution of another constraint, i.e., we allow the definition of auxiliary functions over type variables in constraints. This requirement needs to be captured by the constraint solver accordingly, a description of the constraint solving algorithm is given in Fig. 5.
To deal with constraints whose evaluation depends on the solution of another constraint the algorithm performs a dependency analysis on the given constraint set and partitions the set C such that C_1 contains all the constraints which can be solved directly and C_2 consists of all the constraints depending on the solution of one of the constraints in C_1. Now the constraints in C_1 can be solved gradually and the possibly arisen substitutions are composed with σ and applied to all remaining constraints. Finally, the constraints in C_2 are solved using the algorithm described and the resulting substitution is composed with σ.

5 Designing the Library

Having defined a formalism for automatically deriving type check functionality from a type system's specification we now present the design of a Haskell library utilizing the algorithms of the previous sections and discuss some aspects of the implementation in detail.
To determine the initial feature set of the library we used two languages as case studies and the characteristics of these languages formed the requirements

regarding the expressiveness of our implementation. We chose Mini-ML [3] and FeatherweightJava [14] due to their role as core calculi for pure functional and class-based, object-oriented languages respectively. Additionally their type systems contain interesting typing concepts such as strongly typed expressions without type declarations (type inference) and let-polymorphism in Mini-ML as well as subtyping (via single inheritance), casting, and method override in FeatherweightJava.

The library provides a default abstract syntax for types and the user defines her type system by deduction rules using the given components for abstract syntax and types. Based on a set of typing rules the library's type check function is able to compute the most general type of an expression by interpreting the given inference rules.

5.1 Abstract Syntax

For rapid prototyping of type checking functionality our library ships with a number of default components for abstract syntax and types which fulfill the needed technical requirements to be used in deduction rules.

Typing rules reason at a meta level about the used contexts, expressions, and types. The library's constraint generation function instantiates such rules and replaces all meta-level elements with their object level-pendants. Thus the definitions for expressions and types have to provide object and meta-level versions accordingly. To ease the implementation of abstract syntax suitable for use with our type checking framework we supply common functionality which provides object as well as meta-level versions and fulfills all the technical requirements arising from the constraint generation and the constraint solving algorithms. Some of these components, namely identifiers, sets, sequences, and auxiliary functions in type rules, are presented subsequently.

Names, or to be more precise, identifiers of elements are an essential component of a language's abstract syntax. Given the requirements stated earlier the data type `Identifier` yields everything needed to be used in deduction rules:

```haskell
data Identifier a = MIde String     -- Meta level
                  | Ide  a          -- Object level

type Ide = Identifier String
```

Simple, `String`-based identifiers can be realized by instantiating the type parameter `a` accordingly.

The FeatherweightJava type system depends on sets and sequences over expressions, types and judgements. Consider for example a type rule for method definitions: as part of an inference rule we have to reason about an arbitrary number of parameters at meta level since the exact arity of the method is unknown until the rule is instantiated. To be able to reason about such syntactical elements in deduction rules we introduce data structures representing sets and sequences at object and at meta level:

```haskell
data ISet = MetaISet String
          | ISet [Int]

data Sequence a = MetaSeq ISet a
                | ObjSeq  (Data.Seq.Seq a)

data Set a = MetaSet ISet a
           | ObjSet (Data.Set.Set a)
```

Meta-level sets and sequences over elements e are denotated as the union $\cup_{\mathrm{i}\in\mathrm{I}} \{e_i\}$ and the concatenation $\wedge_{\mathrm{i}\in\mathrm{I}}\, e_i$ of indexed meta-level elements e respectively. This notion is transformed into the HASKELL data types above in a straightforward manner. Meta-level sets and sequences consist of the element e and an index set. This index set can be at meta level, represented by an identifier, or at object level. Object-level index sets are encoded as simple integer lists. Meta-level sets and sequences with an index set at object level can be transformed into object-level sets and sequences by indexing the element e accordingly.

For convenience reasons it is useful to allow auxiliary functions in deduction rules. Such auxiliary functions might be the lookup in a context or the calculation of a class attribute's type. This leads to the question how auxiliary functions, more precisely calls to those functions, can be encoded in order to use them in type rules. Deduction rules reason at meta level over their judgements. Thus potential arguments for auxiliary functions might be at meta level, too. So the function call needs to be deferred until all meta-level arguments are instantiated with a corresponding element at object level. Since HASKELL evaluates lazily, an auxiliary function is not applied to its arguments immediately, but there is no way to change the arguments of such an application thunk afterwards. This problem can be handled by wrapping the function and its arguments in a certain way:

```haskell
data MetaFun b = forall a . (...) => MF (a -> Maybe b) a
```

The call to an auxiliary function is encoded in a simple wrapper data structure containing just the function and its arguments. To deal with varying arity and types the function will be uncurryed and the type variable capturing the arguments is existentially quantified. Note that the wrapped function has to return a `Maybe` value. This allows us to capture the elements of the function's domain for which the function is not defined, i.e., meta-level elements or type variables. In such cases the function returns `Nothing`. This gives us the possibility to determine during constraint solving whether a meta-level function is evaluable or not.

In addition to several components to be used in a language's expression syntax the library supplies an implementation for types which covers a basic set of standard type constructs such as base types, type variables, function and tuple types, type constructors, and type schemes:

```haskell
data Ty = Bottom              -- Bottom type
        | T Ide               -- Base type
        | TV Ide              -- Type variable
        | TF Ty Ty              -- Function type
        | TT [Ty]               -- Tuple type
        | TC Ide [Ty]           -- Type constructor
        | TS [Ty] Ty            -- Type scheme
        | TFun (MetaFun Ty)     -- Type function
        | TSeq (Sequence Ty)    -- Type sequences
        | TSet (Set Ty)         -- Type sets
        | MT String             -- Meta-level type
```

The library's type definition also utilizes all the items presented so far and provides type sets and sequences to be used in deduction rules as well as meta level functions evaluating to a type.

5.2 Inference Rules

As part of our library we enhance the notion for constraint-based deduction rules to allow the user a more convenient way to define her type system. Instead of annotating each judgement with a constraint set and adding the constraints arising from the use of this rule to the conclusion's constraint set, e.g.,

$$\frac{\Gamma \vdash f : T_1 \mid C_1 \qquad \Gamma \vdash e : T_2 \mid C_2}{\Gamma \vdash (f)\, e : T \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to T\}} \tag{\textsc{App}}$$

the arising constraints will be denoted as a premise and the constraint set annotated at each typing judgement will be omitted:

$$\frac{\Gamma \vdash f : T_1 \qquad \Gamma \vdash e : T_2 \qquad T_1 = T_2 \to T}{\Gamma \vdash (f)\, e : T} \tag{\textsc{App}}$$

That is, each constraint given as a premise as well as the constraint sets of the judgement premises will be implicitly added to the conclusion's constraint set. This modification employs a more compact way to define an inference rule even if numerous constraints arise from the use of this rule.

Given this notation, deduction rules can be encoded in a straightforward manner using the following algebraic data types:

```
data Judgement = forall a . (...) => J Context a Ty
               | forall a . (...) => C (Constraint a)

data Rule = Rule { premises   :: [Judgement],
                   conclusion :: Judgement }
```

The data structures for judgements and constraints as well as the algorithms for generating and solving constraints are parametric in the used data type for expressions. This allows the user of our framework to use her own abstract syntax as long as the implementation fulfills certain requirements. These requirements are formulated as type class constraints over the existentially quantified type variable `a` and are omitted for readability reasons in the data type declarations above. We will define some of these requirements throughout the remainder of this section, the omitted ones are merely technical and do not need any further discussion.

In addition to the desired genericity of the library's data structures and algorithms, more complex typing rules might need to reason about different kinds of expressions. To allow the user to define such inference rules, judgements are existentially quantified over their expressions.[1]

5.3 Constraints

Wand's example action table for a simply typed lambda calculus helps us to determine the central constraint domain to be used when formalizing type systems: equality constraints over type expressions. In addition our library supports the use of the logical connectives negation, conjunction, disjunction and implication as well as the use of predicates. Last but not least we allow the user to define solvers for new constraint domains:

```haskell
type Unifier = (Bool, Substitution)

data Constraint a = Eq a a
                  | Not (Constraint a)
                  | And (Constraint a) (Constraint a)
                  | Or  (Constraint a) (Constraint a)
                  | If  (Constraint a) (Constraint a)
                  | Predicate (MetaFun Bool)
                  | Constraint (MetaFun Unifier)
```

Predicates are implemented using an embedded Haskell predicate. User defined constraints are defined in a similar fashion, here the solver for the new constraint domain is supplied as an embedded function.

5.4 Rule Instantiation

The basic technique used for the instantiation of a rule is first-order unification [27]. A rule is instantiable if and only if there exists a most general unifier between the rule's conclusion and the current goal.

Based on a given list of typing rules the constraint generation algorithm tries to instantiate each arising goal with one of the rules. If a matching rule for the current goal has been found the remaining rules are not checked any more. This can be described as a *first-fit-rule-matching semantic* where the rules are implicitly prioritized based on their order in the list. If a matching rule for a subgoal has been found the rule's conclusion can be instantiated by applying the found unifier to it. To complete the instantiation of the rule the premises and constraints of the rule have to be instantiated, too. This task is accomplished

[1] This requirement leads to the problem of defining heterogenous collections in Haskell. Using existential types yields the solution providing the most convenient interface for the user of the library.

in three steps: At first, all substitutions over index sets are applied to the rule's premises and constraints to be able to instantiate and unfold all meta-level sets and sequences contained in those premises and constraints. Secondly, the found unifier is applied to the unfolded judgements. At last, all remaining meta-level types are instantiated with fresh type variables.

This rule instantiation algorithm formulates some of the technical requirements regarding the used abstract syntax. Object and meta-level versions of types and expressions have to be unifiable, the application of substitutions as well as the unfolding of meta-level sets and sequences has to be defined, the instantiation of the remaining meta-level types with fresh type variables has to be stated, and last but not least all evaluable embedded Haskell functions have to be evaluated. These requirements are captured in the type classes `Evaluable`, `Substitutable`, `Instantiable`, and `Unifiable` and the existentially quantified type variables on the data type declarations for judgements, constraints, and meta-level functions are annotated with corresponding type class constraints.

```haskell
class Evaluable a where
    isEvaluable :: a -> Bool
    containsMF  :: a -> Bool
    eval        :: a -> a

class Substitutable a where
    apply :: Substitution -> a -> a

class Instantiable a where
    unfold :: a -> a
    indexM :: a -> Int -> a
    instMT :: a -> TypeCheckM Substitution

class Eq a => Unifiable a where
    unify    :: a -> a -> Unifier
    occursIn :: a -> a -> Bool
```

The type class `Evaluable` provides two discriminator methods `isEvaluable` and `containsMF` for checking whether elements are at object level or if they contain a wrapped up auxiliary function and the method `eval` which evaluates such a meta-level function. The class `Substitutable` captures the application of a substitution to an element and the class `Unifiable` handles unification and occurs-checks. Last but not least, the type class `Instantiable` defines methods for the unfolding of meta-level sets and sequences (`unfold`), the indexing of meta-level elements (`indexM`) as well as the instantiation of all meta-level types contained in an expression. The method `instMT` runs inside a state monad to be able to generate fresh type variables.

Experienced Haskell programmers might object that some of this functionality, e.g. the evaluation of meta-level functions contained in an expression, the application of a substitution to an expression and its children respectively, the unfolding of meta-level sets and sequences, and the instantiation of meta-level types, could have been implemented as part of the library using data type generic programming techniques as described in *Scrap your boilerplate* [18] or *Uniplate* [21] instead of obligating the user to define it by herself. Unfortunately, the use of existential types for meta-level functions and judgements prevents us from using generic programming, since the mentioned libraries can't define generic functions over non-HASKELL-98 data types[2].

Nevertheless our library provides a mechanism to save the user from writing boilerplate code when defining instance declarations for the four type classes stated above. Given some essential properties of the abstract syntax, i.e., a discriminator for meta-level elements, a relation linking meta-level elements to its corresponding object-level pendants, as well as a function handling the indexing of meta-level elements, we can employ HASKELL's compile-time meta-programming facilities [29] to derive the instance declarations for `Evaluable`, `Substitutable`, `Instantiable`, and `Unifiable`. These needed properties are covered by the type class `AST`

```
class AST a where
    index  :: a -> Int -> a
    (~=)   :: a -> a -> Bool
    isMeta :: a -> Bool
```

and by providing an instance declaration for `AST` the user can utilize the library's *Template Haskell* functionality to derive the desired instance declarations for the four type classes capturing the requirements formulated by the constraint generation and the constraint solving function.

5.5 Heterogenous Substitutions

Unification plays an essential role as part of our framework since it is used during rule instantiation to determine the matching rule for an expression and during constraint solving to solve equality constraints. Both the rule instantiation phase and the constraint solver formulate some distinct requirements on the used implementation for substitutions and we want to discuss some of these technical aspects in detail.

Instantiating a rule yields mappings from meta-level contexts to object-level contexts, from meta-level expressions to object-level expressions, and from meta-level types to object-level types. We want to capture all these mappings in one substitution and therefore define substitutions as heterogenous collections of homogenous mappings from meta-level to object-level elements.

As part of our library we implement substitutions as maps from types to collections of homogenous mappings. Since types are not values in HASKELL we use the data type for type representations instead:

```
data S = forall a . S (Map a a)

type Substitution = Map TypeRep S
```

[2] This limitation of nearly all generic programming approaches in the HASKELL universe is discussed in great detail in Alexey Rodriguez Yakushev's PhD thesis [28].

Given those data structures, a substitution can be seen as a collection of key-value pairs where the key element hints on the type the mappings in the value element are ranging over. To capture the heterogenous nature of this collection, the homogenous mappings are existentially quantified again. Basic operations, such as inserting a meta-level/object-level mapping into a substitution, can now be implemented in a straightforward manner:

```haskell
insert :: a -> a -> Substitution -> Substitution
insert k v s = insertWith addKV (typeOf k) kv s
    where
      kv = S (singleton k v)

      addKV _ (S m) =
          case (cast m) :: Maybe (Map a a) of
             Just m' -> S (M.insert k v m')
             _       -> error "Corrupt subst."
```

Based on the type of the mapping we determine the corresponding entry in the underlying map. We have to cast the wrapped up collection of mappings using Haskell's type safe cast operator to be able to insert the given mapping. If mappings over the type `a` are not contained in the map yet, a new entry is added to the map containing just the given mapping.

If we want to apply a substitution to an element we use the same pattern again:

```haskell
apply :: Substitution -> a -> a
apply s x = maybe x (! x) (lookup (typeOf x) s)

(S m) ! x = case (cast m) :: Maybe (Map a a) of
               Just m' -> maybe x id (lookup x m')
               Nothing -> error "Corrupt subst."
```

We determine if a collection of mappings of type `a` exists in the given substitution and lookup the corresponding value. If no mappings over `a` are present or `x` is not stored as a key in the underlying map, `apply s` results to identity.

5.6 Constraint Solving

The constraint solving algorithm described in Sec. 4 performs a dependency analysis on the input and partitions the given constraint set such that constraints containing auxiliary functions depending on the solution of another constraint can be postponed accordingly.

The key question at this point is: how do we determine if an auxiliary function depends on the solution of another constraint? The constraint generation algorithm instantiates meta-level types with fresh type variables and the constraint solver yields a substitution containing mappings from type variables to types. Thus an auxiliary function with type variables as arguments needs to be postponed, if one of these type variables is bound by another constraint, too[3].

[3] To get an idea in which scenarios we benefit from this dependency analysis, the reader might want take a look at the next section where we present an example type

This view on auxiliary functions defines one essential requirement regarding the implementation of the dependency analysis: we need to be able to determine all type variables bound by a constraint. We follow the strategies used so far and define a type class `Vars` which covers the computation of free, bound, and all type variables contained in an element:

```
class Vars a where
    fv :: a -> Set Ty
    fv x  = (vs x) \\ (bv x)

    bv :: a -> Set Ty
    bv x = (vs x) \\ (fv x)

    vs :: a -> Set Ty
    vs x = (fv x) ‘union‘ (bv x)
```

Our library provides instance declarations for all abstract-syntax components described earlier, meta-level functions, types, and constraints. Thus the `Vars` instance declaration for the user's abstract syntax would just define a recursive traversal of the syntactical elements where all type variables are collected. Again, we do not obligate the user to write this boilerplate code and provide *Template Haskell*-functionality to derive the instance declaration automatically.

Being able to select all type variables contained in a constraint the dependency analysis can be implemented in a straightforward manner. Using the `containsMF` method of the `Evaluable` type class we can discriminate all constraints containing an auxiliary function. Given such a constraint we select the type variables of the auxiliary function(s) and calculate the set of type variables bound by all other constraints. If the intersection of these two sets is not empty the auxiliary function might depend on the solution of another constraint and the constraint containing the auxiliary function is postponed for later consideration.

5.7 Error Messages

For real world usage our library has to provide a mechanism to define adequate error messages. Until now, an ill-typed expression yields to the information that a certain constraint could not be solved. This obviously does not qualify as a useful error message and therefore our library provides more elaborate components for error handling.

First of all it has to be stated that constraint-based type rules form an excellent base for defining good error messages. Each constraint defines a consistency condition on the expression(s) a type rule reasons about and the non-solvability of a constraint represents one possible typing error which can occur for this expression. Thus annotating each constraint in an inference rule with an error message covers all possible error cases in a quite convenient way.

system and discuss the topic of type variable arguments in auxiliary functions on an explicit type rule.

Our implementation adapts this idea in a straightforward manner and the data type declaration for constraints is modified such that each constructor is extended with an `ErrorMsg` field accordingly.
A good error message does not only consist of a static message but hints on those expressions which produced the error. Since constraints are defined at meta level, all elements an error message could consider are at meta level, too. Given our implementation for auxiliary functions in deductions rules, error messages can be seen as meta-level functions evaluating to a `String`:

```
data ErrorMsg = ErrorMsg (MetaFun String)
```

This approach yields an easy and straightforward implementation of error messages and fits conveniently in the strategy used so far.

6 A Type Checker for Mini-ML

Mini-ML was developed in [3] in order to present a formal description of the central part of the ML language in natural semantics. Mini-ML is a strongly restricted ML, consisting basically of the simply typed lambda calculus extended with the base types `int` and `bool`, pairs, conditionals and recursive, polymorphic let, and can therefor be seen as a core calculus for functional programming languages.

6.1 Syntax

Since Mini-ML is just an extended, simply Curry-style typed lambda calculus, the syntax definition follows the regular, recursive definition of the untyped lambda calculus in a straightforward manner:

Let $\mathcal{V}$ be a countable set of identifiers (variables), $\mathbb{B} = \{true, false\}$ the set of boolean values and $\mathbb{Z}$ the set of integers. The set of Mini-ML terms is the smallest set $\mathcal{T}_{Mini-ML}$ such that

1. $\mathrm{x} \in \mathcal{T}_{Mini-ML}$ for every $\mathrm{x} \in \mathcal{V}$
2. $\mathrm{b} \in \mathcal{T}_{Mini-ML}$ for every $\mathrm{b} \in \mathbb{B}$
3. $\mathrm{n} \in \mathcal{T}_{Mini-ML}$ for every $\mathrm{n} \in \mathbb{Z}$
4. $\mathrm{e} \in \mathcal{T}_{Mini-ML}$ and $\mathrm{x} \in \mathcal{V} \Rightarrow \lambda\,\mathrm{x.e} \in \mathcal{T}_{Mini-ML}$
5. $\mathrm{e}_1, \mathrm{e}_2 \in \mathcal{T}_{Mini-ML} \Rightarrow (\mathrm{e}_1)\ \mathrm{e}_2 \in \mathcal{T}_{Mini-ML}$
6. $\mathrm{e}_1, \mathrm{e}_2 \in \mathcal{T}_{Mini-ML} \Rightarrow (\mathrm{e}_1, \mathrm{e}_2) \in \mathcal{T}_{Mini-ML}$
7. $\mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3 \in \mathcal{T}_{Mini-ML} \Rightarrow \texttt{if}\ \mathrm{e}_1\ \texttt{then}\ \mathrm{e}_2\ \texttt{else}\ \mathrm{e}_3 \in \mathcal{T}_{Mini-ML}$
8. $\mathrm{e}_1, \mathrm{e}_2 \in \mathcal{T}_{Mini-ML}$ and $\mathrm{x} \in \mathcal{V} \Rightarrow \texttt{let}\ \mathrm{x}\ \texttt{=}\ \mathrm{e}_1\ \texttt{in}\ \mathrm{e}_2 \in \mathcal{T}_{Mini-ML}$
9. $\mathrm{e}_1, \mathrm{e}_2 \in \mathcal{T}_{Mini-ML}$ and $\mathrm{f}, \mathrm{x} \in \mathcal{V} \Rightarrow \texttt{letrec}\ \mathrm{f}\ \texttt{=}\ \lambda\,\mathrm{x.e}_1\ \texttt{in}\ \mathrm{e}_2 \in \mathcal{T}_{Mini-ML}$
10. $\mathrm{e} \in \mathcal{T}_{Mini-ML} \Rightarrow \texttt{fix}\ \mathrm{e} \in \mathcal{T}_{Mini-ML}$

6.2 Encoding

As part of our first example type checker we want to utilize the library's default abstract syntax (see Appendix A) to encode MINI-ML terms and the remainder of this section is used to describe the translation accordingly.

Variables and constants are obviously encoded in their corresponding constructors. All the other syntactical features of MINI-ML are interpreted as combinations of terms and therefore are encoded in a combiner:

Lambda abstraction An untyped lambda abstraction combines a variable with an expression: $\lambda x.\, e \Rightarrow$ `K Abs 2 [Var (Ide``x''), e]`. A typed lambda abstraction (not part of MINI-ML) will be encoded in a similar way, additionally binding a type to the variable: $\lambda x : T.\, e \Rightarrow$ `K Abs 2 [Bind (Ide ``T'') (Var (Ide ``x'')), e]`.

Application An application in the simply typed lambda calculus applies an expression to another one and can therefor be seen as a combination of terms, too: $(f)\, e \Rightarrow$ `K App 2 [f, e]`.

Pairs Being two-ary tuples, pairs combine two MINI-ML expressions: $(e_1, e_2) \Rightarrow$ `K Tuple 2 [e1, e2]`.

Let A let-binding combines a variable with two expressions: `let` $x = e_1$ `in` $e_2 \Rightarrow$ `K Let 3 [Var (Ide ``x''), e1, e2]`. A recursive let is encoded in the same manner, but uses a different tag (`LetRec`).

Conditionals MINI-ML's syntax provides conditionals which range over three expressions and will be encoded in a combiner: `if` c `then` e_1 `else` $e_2 \Rightarrow$ `K IfThenElse 3 [c, e1, e2]`.

6.3 Type system

The type rules for the extended, simply typed lambda calculus presented above will follow standard notations [25]. Since a constraint-based type system is desired, the typing constraints arising in each rule will be explicitly denotated according to the scheme presented in Sec. 5.2.
But before being able to state MINI-ML's type system we need to introduce the notion of type schemes in order to define adequate type rules for polymorphic `let` bindings.

Definition 1.3.1 A type scheme $s = \forall\alpha_1.\ ...\ \forall\alpha_n.\tau$ has a generic instance $t = \forall\beta_1.\ ...\ \forall\beta_n.\tau'$, written $s \succeq t$, if there exists a substitution σ such that

$$\tau' = \sigma(\tau) \text{ with } dom(\sigma) \subseteq \{\alpha_1, ..., \alpha_n\}$$

and all β_i are not free in σ. If s and t are types rather than type schemes, then $s \succeq t$ implies $s = t$.

Definition 1.3.2 To generalize a type over its free type variables we define

$$gen(\varGamma, \tau) = \begin{cases} \forall\alpha_1.\ ...\ \forall\alpha_n.\tau & FV(\tau) \setminus FV(\varGamma) = \{\alpha_1, ..., \alpha_n\} \\ \tau & FV(\tau) \setminus FV(\varGamma) = \emptyset \end{cases}$$

These two auxiliary functions can be implemented in a straight forward manner using the library's default type data structure:

```
(>=) :: Ty -> Ty -> Maybe (Bool, Substitution)
(MT _) >= _      = Nothing
_      >= (MT _) = Nothing
-- check if second type scheme is a
-- generic instance of the first one
(TS tvs1 t1) >= (TS tvs2 t2) =
    let (b1,o) = unify t1 t2
        b2     = fromList (dom o) `isSubsetOf` fromList tvs1
        b3     = null (bv t2 `intersection` fv t1)
    in Just (b1 && b2 && b3, empty)
-- or instantiate type scheme
(TS tvs t1) >= t2 =
    let f tv s = insert tv ((TV . Ide) (freshName "T")) s
        o      = foldr f empty tvs
        t1'    = apply o t1
    in t1' >= t2
t1 >= t2@(TS _ _) = t2 >= t1
-- otherwise, unify types
t1 >= t2 = Just (unify t1 t2)


gen :: Context -> Ty -> Maybe Ty
gen (MCtx _) _ = Nothing
gen _ (MT _)   = Nothing
gen ctx ty  = let tvs  = fv ty `difference` fv ctx
                  tvs' = toList tvs
              in if null tvs then Just ty
                 else Just (TS tvs' ty)
```

Given this notion of type schemes, a constraint-based inference system for Mini-ML expressions can be stated in a straightforward manner as given in Fig. 6.

Having defined a suitable inference system for Mini-ML we now want to use the remainder of this section to describe the encoding of the deduction rules given above using our type checker library. To do so, let us define some meta-level expressions, types, and contexts used commonly in the following type rules:

```
ctx = MCtx "Gamma" ; x  = MIde "x"   ; n  = MConst "n"
e1  = MTerm "e"    ; e1 = MTerm "e1" ; e2 = MTerm "e2"
e3  = MTerm "e3"   ; f  = MTerm "f"  ; t  = MT "T"
t1  = MT "T1"      ; t2 = MT "T2"    ; t3 = MT "T3"
```

Simple Lambda

$$\frac{\Gamma(x) \succeq T}{\Gamma \vdash x : T} \qquad (\textsc{Var})$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \qquad T = T_1 \rightarrow T_2}{\Gamma \vdash \lambda\, x\, .\, e : T} \qquad (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : T_1 \qquad \Gamma \vdash e : T_2 \qquad T_1 = T_2 \rightarrow T}{\Gamma \vdash f\, e : T} \qquad (\textsc{App})$$

Base Types

$$\frac{T = \texttt{Int}}{\Gamma \vdash n : T \;,\; n \in \mathbb{Z}} \qquad (\textsc{Int})$$

$$\frac{T = \texttt{Bool}}{\Gamma \vdash b : T \;,\; b \in \mathbb{B}} \qquad (\textsc{Bool})$$

$$\frac{T = \texttt{Int} \rightarrow (\texttt{Int} \rightarrow \texttt{Int})}{\Gamma \vdash \oplus : T \;,\; \oplus \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/}\}} \qquad (\textsc{Arith})$$

$$\frac{T = \texttt{Int} \rightarrow (\texttt{Int} \rightarrow \texttt{Bool})}{\Gamma \vdash \texttt{==} : T} \qquad (\textsc{Compare})$$

Extensions

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \Gamma \vdash e_3 : T_3 \\ T_1 = \texttt{bool} \qquad T = T_2 \qquad T = T_3\end{array}}{\Gamma \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T} \qquad (\textsc{Cond})$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T = T_1 \times T_2}{\Gamma \vdash (e_1, e_2) : T} \qquad (\textsc{Pair})$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : \mathit{gen}(\Gamma, T_1) \vdash e_2 : T_2 \qquad T = T_2}{\Gamma \vdash \texttt{let}\ x\,\texttt{=}\,e_1\ \texttt{in}\ e_2 : T} \qquad (\textsc{Let})$$

$$\frac{\Gamma \vdash\ \texttt{let}\ x =\ \texttt{fix}\ (\lambda x.e_1)\ \texttt{in}\ e_2 : T_2 \qquad T = T_2}{\Gamma \vdash\ \texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 : T} \qquad (\textsc{Rec-let})$$

$$\frac{\Gamma \vdash f : T_1 \qquad T_1 = T \rightarrow T}{\Gamma \vdash\ \texttt{fix}\ f : T} \qquad (\textsc{Fix})$$

Fig. 6. A constraint-based inference system for Mini-ML

Secondly, we need to lift the two auxiliary functions to the meta level to be able to use them in the to be defined deduction rules, i.e. ,

```
gen ctx ty = TyFun (MF "gen" (uncurry Rule.gen) (ctx,ty))

t1 >= t2 = Constraint (MF ">=" (uncurry Type.>=) (t1,t2))
                      (mkErr "Generic instance check failed."
                         )
```

For all of our type rules we will use a default, non-specific error message:

```
err = mkErr "An error has occurred."
```

A more sophisticated approach for error handling will be described when defining a type checker for FJ in the next section. Given all the needed meta-level elements we now can implement Mini-ML's type system using the data structures of our library.

Variables The typing rule for variable lookup utilizes our notion for calls to auxiliary functions in deduction rules to encode the context lookup (`!`) and the check, if the given type is a generic instance of the inferred one (>=).

```
var :: Rule
var = Rule [ t1 =:= ctx ! Var x <|> err,
             (t1 >= t) <|> err ]
           ( ctx |- Var x <:> t )
```

Lambda Abstraction The typing rule for λ-abstraction uses a meta-level function to insert a typing into the context.

```
abs :: Rule
abs = Rule [ ctx' |- e <:> t2 , t =:= (t1 ->: t2) <|> err ]
           ( ctx |- K Abs 2 [Var x, e] <:> t )
      where
        ctx' = mInsertCtx (Var x) t1 ctx
```

Application The inference rule for applications can be encoded in a straightforward manner using our library's combinators.

```
app :: Rule
app = Rule [ ctx |- f <:> t1 , ctx |- e <:> t2,
             t1 =:= (t2 ->: t) <|> err ]
           ( ctx |- (K App 2 [f,e]) <:> t )
```

Base Types The deduction rules for the base types `int` and `bool` as well as the build-in functionality on these two types can be defined as following:

```
int  = mkT "int"
bool = mkT "bool"

true = Rule [ t =:= bool <|> err ]
            ( ctx |- Var (Ide "true") <:> t )

false = Rule [ t =:= bool <|> err ]
             ( ctx |- Var (Ide "false") <:> t )

const = Rule [ t =:= int <|> err ]
             ( ctx |- n <:> t )

add = Rule [ t =:= (int ->: (int ->: int)) <|> err ]
           ( ctx |- Var (Ide "+") <:> t )

sub = Rule [ t =:= (int ->: (int ->: int)) <|> err ]
           ( ctx |- Var (Ide "-") <:> t )

mul = Rule [ t =:= (int ->: (int ->: int)) <|> err ]
           ( ctx |- Var (Ide "*") <:> t )

div = Rule [ t =:= (int ->: (int ->: int)) <|> err ]
           ( ctx |- Var (Ide "/") <:> t )

eqi = Rule [ t =:= (int ->: (int ->: bool)) <|> err ]
           ( ctx |- Var (Ide "==") <:> t )
```

Conditionals The inference rule for conditionals can be encoded in a straightforward manner using our library's combinators.

```
cond :: Rule
cond = Rule [ ctx |- e1 <:> t1, ctx |- e2 <:> t2
            , ctx |- e3 <:> t3, t1 =:= bool <|> err
            , t =:= t2 <|> err, t =:= t3 <|> err ]
            ( ctx |- (K IfThenElse 3 [e1,e2,e3]) <:> t )
```

Pairs The typing rule for pairs can again be encoded in a quite elegant way using our library's combinators.

```
pair :: Rule
pair = Rule [ ctx |- e1 <:> t1 , ctx |- e2 <:> t2,
              t =:= (t1 ** t2) <|> err ]
            ( ctx |- (K Tuple 2 [e1,e2]) <:> t )
```

Let-Bindings The typing rules for polymorphic and recursive let-bindings utilize our notion for calls to auxiliary functions in deduction rules to encode the insertion of a typing into a context as well as the generation of a type scheme.

```
letpoly :: Rule
letpoly = Rule [ ctx |- e1 <:> t1, t2 =:= gen ctx t1 <|> err
               , mInsertCtx (Var x) t2 ctx |- e2 <:> t3
               , t =:= t3 <|> err ]
               ( ctx |- (K Let 3 [Var x,e1,e2]) <:> t )


letrec :: Rule
letrec = Rule [ ctx |- (K Let 3 [Var x, fix, e2]) <:> t2,
                t =:= t2 <|> err ]
              ( ctx |- (K LetRec 3 [Var x, e1, e2]) <:> t )
         where
           fix = K Fix 1 [K Abs 2 [Var x, e1]]


fix :: Rule
fix = Rule [ ctx |- f <:> t1 , t1 =:= (t ->: t) <|> err ]
           ( ctx |- (K Fix 1 [f]) <:> t )
```

6.4 Type Checker

Having encoded Mini-ML's type system using the data structures of our library we know can derive the desired type checker by defining the functions `computeType` and `checkType` with the help of our library. In this example we use the default type check mode which just employs an empty initial context on the constraint solver.

```
mlRules = [ true, false, const, add, sub , mul, div, eqi
          , and, or, var, abs, app, cond, pair, letpoly
          , letrec, fix ]

computeType :: Term -> TypeCheckResult Ty
computeType exp = computeTy defaultMode mlRules exp

checkType :: Term -> Ty -> TypeCheckResult Ty
checkType exp ty = checkTy defaultMode mlRules exp ty
```

Note that in this example we also could have used a non-empty initial context which contains the typings for the build-in arithmetic and comparison functions. In this case we could have omitted the corresponding type rules.

7 A Type Checker for FJ

FeatherweightJava [14] was introduced by Igarashi, Pierce, and Wadler in order to present a lightweight version of Java which enables rigorous arguments about key properties such as type safety. The language omits almost all features of the full Java to obtain a small calculus for which detailed proofs of type safety become considerably easy.

Nevertheless, it still captures the essential computational “feel”, providing classes, methods, fields, inheritance and dynamic typecasts with a semantic closely following Java’s. In this sense, every FJ program is an executable Java program.

7.1 Syntax

The syntax of FJ therefore is equivalent to Java’s syntax with following omissions: concurrency (threads), inner classes, reflection, assignment, interfaces, overloading, messages to super, null pointers, base types (int, boolean, etc.), abstract method declarations, shadowing of superclass fields by subclass fields, access control (public, private, etc.) and exceptions. The features of Java that FJ does model include mutually recursive class definitions, object creation, field access, method invocation, method override, method recursion via *this*, subtyping and casting.

An EBNF-style syntax definition of FJ is given below. The nonterminal C ranges over class names, f ranges over field names, m ranges over method names and x ranges over all valid Java identifier.

$$\begin{array}{lll}
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\ [C\ f\ ;]^*\ K\ M^*\ \} \\
K & ::= & C\ \left([C\ f]^*\right)\ \{\ \texttt{super}\ \left(f^*\right)\ ;\ [\texttt{this}\ .\ f = f]^*\ \} \\
M & ::= & C\ m\ \left([C\ x]^*\right)\ \{\ \texttt{return}\ e\ ;\ \} \\
e & ::= & x\ \mid e\ .\ m\ \left(e^*\right)\ \mid \texttt{new}\ C\ \left(e^*\right)\ \mid (C)\ e
\end{array}$$

As part of this example we want to define abstract syntax for FJ suitable to be used within our framework. To do so, we define data structures for classes, methods, constructors, and expressions, but add according meta-level variants to each type additionally:

```haskell
-- FJ expressions
data FJExpr = Var Ide
            | Invoke FJExpr FJExpr
            | Meth Ide (Sequence FJExpr)
            | New Ty (Sequence FJExpr)
            | Cast Ty FJExpr
            | Assign FJExpr FJExpr
            | Return FJExpr
            | MExpr String  -- Meta level expression

```

```
-- FJ method declarations
data FJMethod = M { mRetTy  :: Ty
                  , mName   :: Ide
                  , mParams :: Sequence (Ty, FJExpr)
                  , mBody   :: FJExpr
                  }
              | MM String  -- Meta level method


-- FJ constructor declaration
data FJConstructor = C { cName   :: Ty
                       , cParams :: Sequence (Ty, FJExpr)
                       , super   :: FJExpr
                       , assigns :: Sequence FJExpr
                       }
                   | MC String  -- Meta level constructor


-- FJ class definition
data FJClass = Class { clName      :: Ty
                     , superClass  :: Ty
                     , attributes  :: Sequence (Ty,FJExpr)
                     , constructor :: FJConstructor
                     , methods     :: Sequence FJMethod
                     }
```

Secondly, we give instance declarations for the type class AST by providing a function for indexing meta-level elements, a relation mapping all meta-level elements to their corresponding object-level pendants, and a discriminator. We omit the instance declarations for FJMethod, FJConstructor, and FJClass, their implementation follows the exact same pattern as the one for FJExpr.

```
instance AST FJExpr where
    index (Var x)       n = Var (index x n)
    index (Invoke e f)  n = Invoke (index e n) (index f n)
    index (Meth m args) n = Meth (index m n) (indexM args n)
    index (New c args)  n = New (index c n) (indexM args n)
    index (Cast c e)    n = Cast (index c n) (index e n)
    index (Assign l r)  n = Assign (index l n) (index r n)
    index (Return e)    n = Return (index e n)
    index (MExpr ide)   n = MExpr (ide ++ show n)

    (MExpr _) ~= e = case e of
                       MExpr _ -> False
                       _       -> True
    _ ~= _         = False

    isMeta (MExpr _) = True
    isMeta _         = False
```

Now we can utilize the library's *Template Haskell* based functionality to derive code for the evaluation of the meta-level functions, the application of substitutions, the instantiation of meta-level sets and sequences, the unification of object and meta-level elements, and the collection of type variables contained by an element.

```
$(deriveEvaluable ''FJExpr)

$(deriveSubstitutable ''FJExpr)

$(deriveInstantiable ''FJExpr)

$(deriveUnifiable ''FJExpr)

$(deriveVars ''FJExpr)
```

Finally our abstract syntax for FJ classes and expressions is ready to be used within our type checking framework.

7.2 Type rules

Having defined FJ's syntax it becomes clear that we need to reason about possibly empty sets and sequences of expressions (e.g. parameter lists), fields, methods, and even judgements and constraints in the type rules. Since deduction rules reason at meta level about their premises and conclusion, the need for a concise notion for sets and sequences at meta level arises. We follow standard notion used in set theory and define the combinators

$$\bigwedge_{i \in I} e_i := e_1 \ e_2 \ \dots \ e_{|I|} \quad \text{and} \quad \bigcup_{i \in I} e_i := \{\, e_1, \, \dots \, , e_{|I|} \,\}.$$

Explicitly annotating the index set I at each combinator allows us to reason about the same sequences/sets in different premises. If only one index set is used throughout all premises of a typing rule it might be omitted.

As part of our definition of a constraint-based type system for FJ we will utilize some auxiliary functions providing commonly needed functionality (like calculating the fields of a class or the type of a method) in the deduction rules:

Field lookup:

$$\mathit{fields}(\text{Object}) = \emptyset$$

$$\frac{CT(C) = \text{class } C \text{ extends } D \ \{ \ \bigwedge_{i \in I} (C_i \ f_i;) \ K \ \bigwedge_{j \in J} M_j \ \}}{\mathit{fields}(C) = \bigcup_{i \in I} \{C_i \ f_i\} \ \cup \mathit{fields}(D)}$$

The lookup of a class' fields can be implemented in a straightforward manner by folding over the fields of the desired class and its super classes.

```
fields :: [FJClass] -> Ty -> Maybe (Set (Ty,FJExpr))
fields _  (TV _)               = Nothing
fields [] (T (Ide "Object"))  = Just $ ObjSet (Data.Set.empty)
fields [] _                    = Nothing
fields prog c =
    Just $ foldl
      (\ s1 s2 -> fromJust (union s1 s2)) emptySet
      (mapMaybe (flds prog) (c:(superClasses prog c)))

flds _ (T (Ide "Object")) = Just $ ObjSet (Data.Set.empty)
flds [] c       = Just $ ObjSet (Data.Set.empty)
flds (cl:cls) c = if clName cl == c
                  then Just $ setFromSeq (attributes cl)
                  else flds cls c
```

Field type lookup:

$$ftype(f, C) = \begin{cases} \bullet & \mid C = Object \\ D & \mid D\ f \in fields(C) \end{cases}$$

The implementation of the *ftype* function just looks up if the given field exists in the given class and returns its type accordingly.

```
ftype :: [FJClass] -> FJExpr -> Ty -> Maybe Ty
ftype _ _ (TV _)               = Nothing
ftype _ _ (T (Ide "Object"))  = Nothing
ftype prog f c = if isNothing cls then Nothing
                 else ty
    where
      cls            = find (\ cl -> clName cl == c) prog
      (ObjSeq as)   = attributes $ fromJust cls
      ty             = lookup f $ map swap (toList as)
```

Method type lookup:

$$\frac{\begin{array}{c} CT(C) = \text{class } C \text{ extends } D\ \{\ \bigwedge_{j \in J} (C_j\ f_j;)\ K\ \bigwedge_{l \in L} M_l\ \} \\ B\ m(\bigwedge_{i \in I} B_i\ x_i)\ \{...\} \in \bigwedge_{l \in L} M_l \end{array}}{mtype(m, C) = \bigwedge_{i \in I} B_i \to B}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\ \bigwedge_{j \in J} (C_j\ f_j;)\ K\ \bigwedge_{l \in L} M_l\ \} \qquad m \notin \bigwedge_{l \in L} M_l}{mtype(m, C) = mtype(m, D)}$$

The lookup of a method's type can be implemented in the same style as the lookup of a field's type:

```haskell
mtype :: [FJClass] -> Ide -> Ty -> Maybe Ty
mtype _ _ (TV _)              = Nothing
mtype _ _ (T (Ide "Object"))  = Just Bottom
mtype prog m c = if isNothing cls then Just Bottom
                 else ty
    where
      cls       = find (\ cl -> clName cl == c) prog
      ObjSeq ms = methods (fromJust cls)
      mth       = find (\ mth -> mName mth == m) (toList ms)
      ty        = maybe (Just Bottom) (Just . mkMT) mth

mkMT :: FJMethod -> Ty
mkMT (M rt _ (ObjSeq ps) _) = (extractTSeq ps) ->: rt
    where
      extractTSeq = (TySeq . ObjSeq) . (fmap fst)
```

Super class lookup:

$$superClass(C) = \begin{cases} \bullet & \mid C = Object \\ D & \mid CT(C) = \text{class } C \text{ extends } D \text{ \{ ...\}} \end{cases}$$

This function can be implemented straightforward by just looking up the given class and selecting its superclass.

```haskell
superCls :: [FJClass] -> Ty -> Maybe Ty
superCls _ (TV _)              = Nothing
superCls _ (T (Ide "Object"))  = Just Bottom
superCls [] _                  = Just Bottom
superCls (c:cls) cl = if clName c == cl
                      then Just (superClass c)
                      else superCls cls cl
```

Given these auxiliary functions and a notion for sets and sequences at meta level we now have everything at hand to define a constraint-based type system for FJ in Fig. 7 and Fig. 8.

The constraint-based typing rules for FJ expressions are essentially adapted from the original formalization given in [14]. Changes have been made to the Field typing rule, where the type of a field is determined with the help of an auxiliary function (which is similar to the way a method's type is determined).

The rules for classes and methods needed much stronger modifications: First of all, an additional type rule for constructors is introduced. This rule makes sure the constructor is syntactically correct according to the FJ class specification. Thus, this functionality is removed from the original Class rule, which now just generates constraints verifying that constructor and methods are "Ok" in this class. The Method typing rule follows the original definition, except that the super class of the class which encapsulates the method is determined with

Method typing:

$$\frac{\begin{array}{c}(\bigcup\limits_{j\in I} x_j : C_j), \mathit{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \qquad \mathit{superClass}(C) = D \\ \text{if } \mathit{mtype}(m, D) = (\bigwedge\limits_{k\in I} D_k) \rightarrow D_0, \text{ then } (\bigwedge\limits_{l\in I} C_l = D_l) \text{ and } C_0 = D_0 \\ T = \mathit{Ok\ in\ C}\end{array}}{\Gamma \vdash C_0\ m(\bigwedge\limits_{i\in I} C_i\ x_i)\ \{\ \text{return } e_0\ ;\ \} : T} \ (\textsc{Method})$$

Constructor typing:

$$\frac{\begin{array}{c}\mathit{superClass}(C) = D \qquad \mathit{fields}(C) \backslash \mathit{fields}(D) = \bigcup\limits_{n\in K} \{E_n\ f_n\} \\ \mathit{fields}(C) = \bigcup\limits_{l\in I} \{C_l\ x_l\} \qquad \mathit{fields}(D) = \bigcup\limits_{m\in J} \{D_m\ x_m\} \\ T = \mathit{Ok\ in\ C}\end{array}}{\Gamma \vdash C\ (\bigwedge\limits_{i\in I} C_i\ x_i)\ \{\ \mathit{super}(\bigwedge\limits_{j\in J} x_j);\ \bigwedge\limits_{k\in K} (\mathit{this}.f_k = x_k;)\ \} : T} \ (\textsc{Constructor})$$

Class typing:

$$\frac{\Gamma \vdash K : \mathit{Ok\ in\ C} \qquad \bigwedge\limits_{k\in J} (\Gamma \vdash M_j : \mathit{Ok\ in\ C}) \qquad T = \mathit{Ok}}{\Gamma \vdash \text{class } C \text{ extends } D\ \{\ \bigwedge\limits_{i\in I} (C_i\ f_i;)\ K\ \bigwedge\limits_{j\in J} M_j\ \} : T} \qquad (\textsc{Class})$$

Fig. 7. FJ class, method and constructor typing.

the help of an auxiliary function. For this typing rule an additional note on the (possibly not that intuitive) conditional constraint should be given: This constraint defines FJ's functionality to override base class methods in a sub class, i.e., if a sub class uses the same name for one of its methods as used in its base classes, then it must override this method by using the same parameter and return types.

Expression typing:

$$\frac{T = \Gamma(x)}{\Gamma \vdash x : T} \qquad \text{(Var)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad T = \mathit{ftype}(f, C_0)}{\Gamma \vdash e_0.f : T} \qquad \text{(Field)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad \bigwedge_{j \in I} (\Gamma \vdash e_j : C_j) \quad \mathit{mtype}(m, C_0) = (\bigwedge_{k \in I} D_k) \to C \quad \bigwedge_{l \in I} (C_l <: D_l) \qquad T = C}{\Gamma \vdash e_0.m(\bigwedge_{i \in I} e_i) : T} \qquad \text{(Invk)}$$

$$\frac{\bigwedge_{j \in I} (\Gamma \vdash e_j : C_j) \quad \mathit{fields}(C) = \bigcup_{k \in I} \{D_k\ f_k\} \quad \bigwedge_{l \in I} (C_l <: D_l) \qquad T = C}{\Gamma \vdash \texttt{new}\ C(\bigwedge_{i \in I} e_i) : T} \qquad \text{(New)}$$

$$\frac{\Gamma \vdash e : D \qquad D <: C \qquad T = C}{\Gamma \vdash (C)\, e : T} \qquad \text{(U-Cast)}$$

$$\frac{\Gamma \vdash e : D \qquad C <: D \qquad C \neq D \qquad T = C}{\Gamma \vdash (C)\, e : T} \qquad \text{(D-Cast)}$$

Fig. 8. FJ expression typing.

Having defined a set of constraint-based inference rules to cover the FJ type system we now can start to encode these rules using the data structures of our library. To do so, we define the needed meta-level expressions, types, contexts, identifier, methods, and contexts again:

```
ctx = MCtx "Gamma" ; m = MIde "m" ; m_j = MM "M" ; k = MC "K"

this = Var (Ide "this") ; f = Var (MIde "f")
x    = Var (MIde "x")    ; f_i = Var (MIde "f")
x_i  = Var (MIde "x")    ; f_j = Var (MIde "f")
x_j  = Var (MIde "x")    ; f_k = Var (MIde "f")
x_k  = Var (MIde "x")
e = MExpr "e" ; e0 = MExpr "e0" ; e_i = MExpr "e"

c   = MT "C"  ; d   = MT "D"  ; c'  = MT "C'"
c0  = MT "C0" ; d0  = MT "D0" ; ec  = MT "E"
c_i = MT "C"  ; d_i = MT "D"  ; e0C = MT "E0"
c_j = MT "C"  ; d_j = MT "D"  ; t   = MT "T"
```

```
c_k = MT "C"  ; d_k = MT "D"  ;
```

Secondly, we need to lift all auxiliary functions to the meta level to be able to use them in the to be defined deduction rules, e.g.,

```
fields :: [FJClass] -> Ty -> Set (Ty,FJExpr)
fields prog c =
    SetFun (MF "fields" (uncurry Auxiliary.fields) (prog,c))
```

For the subtype relation we define a pair of operators yielding a convenient notion for defining subtype constraints:

```
(==>) :: [FJClass] -> (Ty, Ty) -> Constraint Ty
prog ==> (s,t) =
    Predicate (MF "<:" (uncurry (subtype prog)) (s,t))
    (mkErr "Error: could not solve subtype constraint.")

s <: t = (s,t)

infix 6 <:
infix 5 ==>
```

All other auxiliary functions are lifted in the same way to the meta level as given in the `fields` example, we omit the corresponding Haskell code.
Given the meta-level elements defined above we now have everything at hand to encode the constraint-based type system for FJ using the data structures of our library.

Variables

```
var :: Rule
var = Rule [ t =:= (ctx ! x) <|> err ]
           ( ctx |- x <:> t )
    where
      err = varError ctx x t
```

The inference rule for variables is encoded in a straightforward manner, except that this time we employ more sophisticated error handling. Instead of printing just a static message in case solving the rule's sole equality constraint fails, we define a function producing a message hinting on the variable as well as the error reason. If the variable is not contained in the given context, this variable is not defined in the current scope and a corresponding error message is generated. Otherwise, the inferred type for the variable does not match the type given in the conclusion's context and a message hinting on the inferred type as well as the expected type is generated. Last but not least, the function generating the error messages is lifted to the meta level and can now be attached to a constraint:

```
varError :: Context -> FJExpr -> Ty -> ErrorMsg
varError ctx x ty =
    ErrorMsg x (MF "" (Just . msg) (ctx,x,ty))
```

```
msg :: (Context, FJExpr, Ty) -> String
msg (ctx,x,ty) =
    case ((ctx ! x) :: Maybe Ty) of
      Nothing  -> "Undefined variable: " ++ pprint x
      Just ty’ -> "Type Missmatch: " ++
                  "Could not match expected type ‘" ++
                  pprint ty ++ "’ against inferred type ‘" ++
                  pprint ty’ ++ "’"
```

Field access

Defining the type rule for accessing an object’s fields comes easy using the corresponding auxiliary function presented earlier. Again, we define an explicit error message handling all the cases which can occur when solving the equality constraint fails.

```
field :: [FJClass] -> Rule
field prog = Rule [ ctx |- e <:> d
                  , c =:= ftype prog f d <|> err ]
                  ( ctx |- Invoke e f <:> c )
    where
      err = fieldError prog (Invoke e f) f c d
```

```
fieldError p exp f c d =
    ErrorMsg exp (MF "" (Just . msg) (p,f,c,d))

msg :: ([FJClass],FJExpr,Ty,Ty) -> String
msg (p,f,c,d) =
    case (ftype p f d) of
      Nothing -> pprint f ++ " is not a field of class " ++
                 pprint d
      Just c’ -> "Type Missmatch: " ++
                 "Could not match expected type ‘" ++
                 pprint c ++ "’ against inferred type ‘" ++
                 pprint c’ ++ "’"
```

Method invocation

As part of the definition of the deduction rule covering method invocations we utilize the library’s convenience function `mSeq` to define meta-level sequences in a more compact way. Additionally, we introduce a generic error message handling type mismatch errors and use it to produce a useful error message if the inferred and the expected return type of the method do not match. The error messages for the constraint covering the method’s arity and its parameter types are just static messages again. We omit more sophisticated error handling in these cases to keep the presentation of the rule a bit more compact. The earlier examples showed how to define useful error messages with the help of our library. For the rest of this section we will keep the error messages short and simple and use just static strings. Since the upcoming rule definitions are more complex than the ones presented so far, this simplification is expected to help to keep the encoded rules more compact by allowing us to focus on the interesting parts of the rule encodings.

```
invoke prog =
    Rule [ ctx |- e <:> ec
         , mSeq "I" (ctx |- e_i <:> c_i)
         , mtype prog m ec =:= (mSeq "I" d_i ->: d) <|> err1
         , mSeq "I" ((prog ==> c_i <: d_i) <|> err2)
         , c =:= d <|> err3 ]
         ( ctx |- Invoke e mcall <:> c )
        where
          mcall = Meth m (mseq "I" e_i)
          err1  = mkErr "Wrong number of arguments."
          err2  = mkErr "Wrong argument types."
          err3  = typeMissmatch (Invoke e mcall) c d
```

```
typeMissmatch expr t1 t2 =
    ErrorMsg expr (MF "" (Just . msg) (t1,t2))

msg (t1,t2) =
    "Type Missmatch: Could not match expected type ‘"
    ++ pprint t1 ++ "’ against inferred type ‘" ++
    pprint c’ ++ "’"
```

Constructor calls

The type rule for constructor calls follows the one for method invocations closely. Note the difference between the two convenience functions `mSeq` and `mseq`. While the latter one is just an abbreviation for a meta-level sequence over a meta-level index set, `mSeq` is an overloaded function wrapping such a meta-level sequence in another constructor, in this case a sequence of judgements (`JSeq`) and a sequence of constraints (`ConstraintSeq`).

```
new prog =
    Rule [ mSeq "I" (ctx |- e_i <:> c_i)
         , fields prog c =:= mset "I" (d_i, f_i) <|> err1
         , mSeq "I" ((prog ==> c_i <: d_i) <|> err2)
         , d =:= c <|> err3 ]
         ( ctx |- New c (mseq "I" e_i) <:> d )
    where
      err1 = mkErr "Wrong constructor call."
      err2 = mkErr "Wrong argument types."
      err3 = typeMissmatch (New c (mseq "I" e_i)) d c
```

Up and down casts

The type rules for up and down casts need some special treatment as part of their encoding in the library's data structures. The *first-fit-rule-matching* semantics of our rule instantiation algorithm forbids the use of rules with overlapping conclusions. Luckily, the two rules for up and down casts can be easily merged into one rule by combining the arising constraints in a disjunction.

```
cast prog =
    Rule [ ctx |- e <:> d
         , Or upcast downcast <|> mkErr "Cast failed."
         , ec =:= c <|> typeMissmatch (Cast c e) ec c ]
         ( ctx |- Cast c e <:> ec )
    where
      upcast   = prog ==> d <: c
      downcast = And (prog ==> c <: d) (c =/= d)
```

Method definitions

The type rule for method definitions introduces a conditional constraint to cover the overriding facilities of FJ. If a method definition overrides the definition of a method in one of its super classes, all parameter types must be subtypes of the overridden method's parameters. All other premises and constraints are encoded in a similar fashion as the ones presented so far.

```
method prog =
    Rule [ c' =:= OkIn c <|> mkErr "Unification error."
         , ctx' |- e0 <:> e0C
         , (prog ==> e0C <: c0) <|>
           mkErr "Wrong return type in method body."
         , superClass prog c =:= d <|>
           mkErr "Superclass lookup failed."
         , If override (And subtypes (c0 =:= d0)) <|>
           mkErr "Wrong override." ]
          ( ctx |- M c0 m params (Return e0) <:> c' )
    where
      params   = mseq "I" (c_i,x_i)
      ctx'     = insertCtx this c (ctxFromSeq params)
      override = mtype prog m d =:= (mSeq "I" d ->: d0)
      subtypes = mSeq "I" (prog ==> c_i <: d_i)
```

Constructor definitions

The type rule for constructor definitions basically covers the syntactical requirements formulated by the FJ language definition, i.e., all class fields are initialized by the constructor parameters with the help of the super constructor and assignments. The used auxiliary function \\ denotes set difference lifted to the meta level and is already defined as part of the library.

```
constructor prog =
    Rule [ superClass prog c =:= d <|>
           mkErr "Superclass lookup failed."
         , fields prog c =:= mset "I" (c_i,x_i) <|>
           mkErr "Wrong number of parameters."
         , fields prog d =:= mset "J" (d_j,x_j) <|>
           mkErr "Wrong number of args in super call."
         , flds =:= mset "K" (e_k,f_k) <|>
           mkErr "Wrong number of assignments."
         , t =:= OkIn c <|> mkErr "Type Missmatch." ]
         ( ctx |- (C c ps s as) <:> t )
    where
      ps   = mseq "I" (c_i,x_i)
      s    = Meth (Ide "super") (mseq "J" x_j)
      as   = mseq "K" (Assign (Invoke this f_k) f_k)
      e_k  = MT "E"
      flds = fields prog c \\ fields prog d
```

Class definitions

The type rule for class definitions just makes sure the constructor definition as well as all method definitions are defined correctly in this class.

```
fjclass prog = Rule [ ctx |- k <:> okInC
                    , mSeq "J" (ctx |- m_j <:> okInC)
                    , t =:= mkT "Ok" <|> mkErr "Type missmatch." ]
                    ( ctx |- (Class c d as k ms) <:> t )
    where
      as     = mseq "I" (c_i,f_i)
      ms     = mseq "J" m_j
      okInC  = OkIn c
```

Using the encoded rules a type checker for FJ can be derived in the same style as presented in the previous section.

8 Visualization

To support the user of our library during the development of the type checking phase, we implemented functionality to visualize the constraint generation and the constraint solving process. Using this visualization tool helps the developer to understand the derived type checker in a more abstract way by allowing him to trace the type derivation and the used solvers in a fine grained manner.

To implement such a tool a sufficient framework for graphical user interfaces was needed. We wanted to implement this tool in HASKELL, so the library could be used directly without calls from another language like C, thus a HASKELL binding for this toolkit was necessary. The second important criterion was cross-platform support.

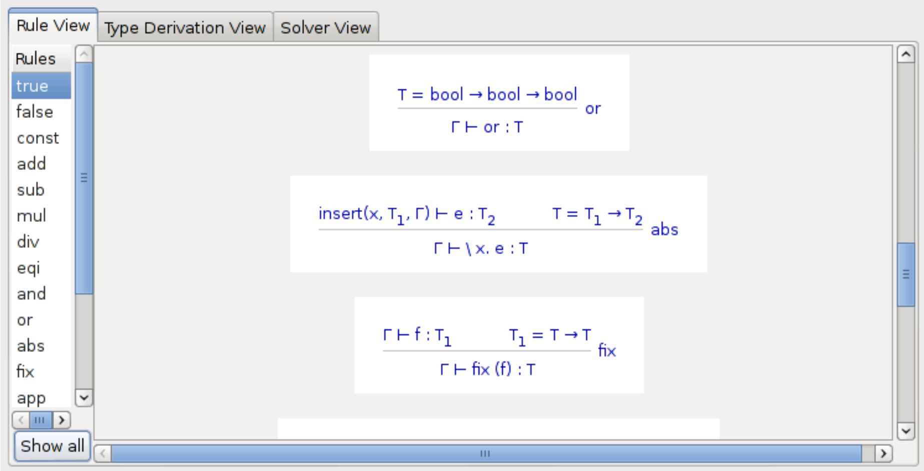

Rule View
Type Derivation View
Solver View
Rules
true
false
const
add
sub
mul
div
eqi
and
or
abs
fix
app
Show all
$\frac{T = \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}}{\Gamma \vdash \text{or} : T}$ or
$\frac{\text{insert}(x, T_1, \Gamma) \vdash e : T_2 \qquad T = T_1 \rightarrow T_2}{\Gamma \vdash \backslash x.\ e : T}$ abs
$\frac{\Gamma \vdash f : T_1 \qquad T_1 = T \rightarrow T}{\Gamma \vdash \text{fix}\ (f) : T}$ fix

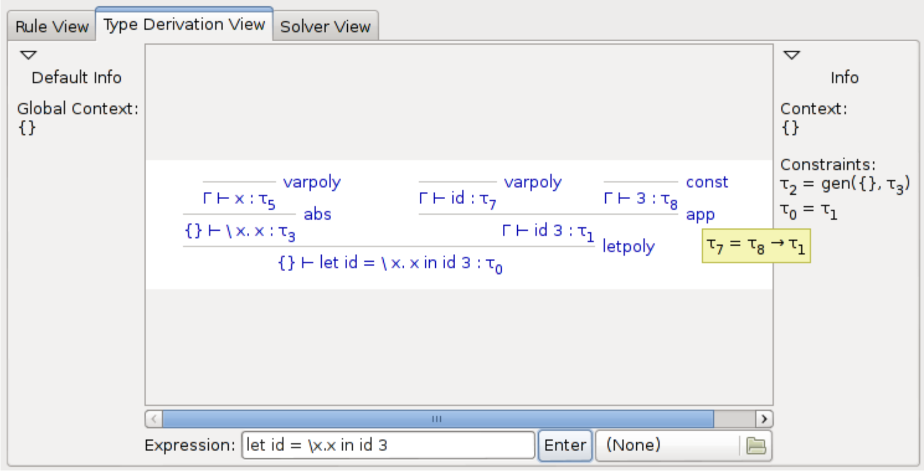
Fig. 9. Rule View

Since the number of well documented, stable bindings of gui frameworks for Haskell that work cross-platform is rather limited, the choice came down to *wxHaskell (wxWidgets)* [34] and *Gtk2Hs (Gtk)* [8]. Both frameworks are mature and widely used. *Gtk* provides the gui interface builder *Glade*, which tipped the favor towards *Gtk2Hs* to implement the graphical user interface of the tool.

As part of the tool design we identified three main work flows which should be supported accordingly. At first, the user should be able to get a concise overview of the typing rules he implemented using the library's EDSL, thus the tool must provide a sufficient rendering of the defined deduction rules. Secondly, the tool should allow the user to trace the library's constraint generation algorithm by rendering the derivation tree for a given expression and thus giving the user the opportunity to understand which rules have been instantiated and which constraints have been collected. Last but not least the tool should allow the user to trace the constraint solving process step by step.

To capture these requirements we designed three main views within the tool: the *Rule View*, the *Type Derivation View*, and the *Solver View*. The *Rule View* provides a "pretty" rendering of the user's type system, the *Type Derivation View* renders the type derivation tree of a given expression, and the *Solver View* is basically a step-by-step trace of the used solvers.

The graphical user interface of the tool was developed with the help of *Gtk*'s gui designer *Glade*. To gather all information needed for the visualization of the different phases, the constraint generation and the constraint solving algorithms had to be adapted slightly. The core algorithms are still the same but the generation and solving process is documented with additional informations in separate data structures for the visualization purpose. The constraint generation algorithm returns a plain list of constraints, neglecting the tree structure which



Global Context:
{}

Context:
{}

Constraints:
$\tau_2 = \mathrm{gen}(\{\}, \tau_3)$
$\tau_0 = \tau_1$

$\tau_7 = \tau_8 \rightarrow \tau_1$

$\Gamma \vdash x : \tau_5$

$\{\} \vdash \backslash x.\, x : \tau_3$

$\Gamma \vdash id : \tau_7$

$\Gamma \vdash 3 : \tau_8$

$\Gamma \vdash id\ 3 : \tau_1$

$\{\} \vdash let\ id = \backslash x.\, x\ in\ id\ 3 : \tau_0$

Expression: let id = \x.x in id 3

(None)

Fig. 10. Type Derivation View

reflects the dependencies and order of the generation process. But since that is exactly what we want to show in the *Type Derivation View* the implementation now returns the normal list but also a tree in which all these informations are stored.

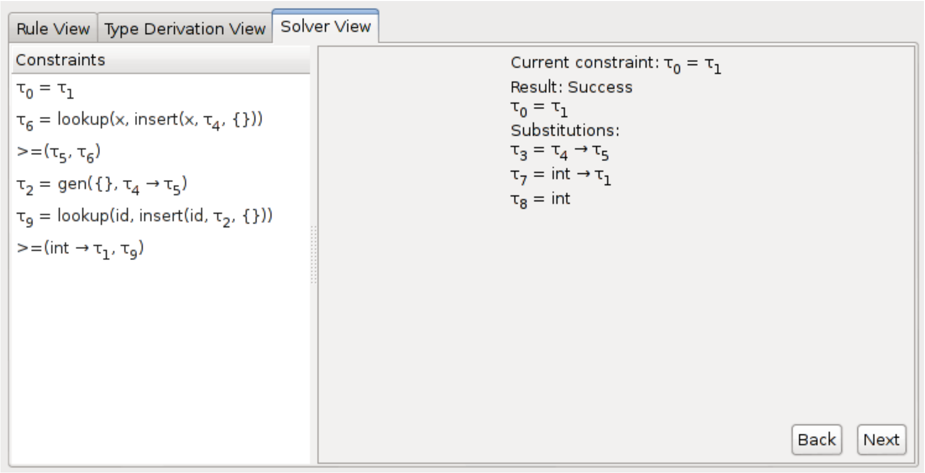
In the implementation of the constraint solving algorithm we introduced a data structure which stores in each solving step the current constraint, the solution of that constraint, the accumulated result substitution, and the constraints jet to be solved:

```haskell
data IntermediateResult =
    IRes { con    :: Judgement     -- current constraint
         , res    :: Result        -- solution of constraint
         , subs   :: Substitution  -- acc. result substitution
         , unsolv:: [Judgement]    -- remaining constraints
         }
```

A list of these intermediate results is than returned along the proper result allowing to trace step-by-step through the constraint solving process.

With these modifications we are able to visualize the inner machinery of our framework. To get a better feeling, what such a derived tool could look like, let us take a closer look on the gui-based inspector tool derived for our MiniML example.

The *Rule View* (Fig. 9) displays the constraint-based deduction rules defined as part of MiniML's inference system in a more readable manner. The left side of the view is a list, where a rule can be selected to be displayed on the right. Under that list is a button to display all rules, one below the other, on the right side.



Constraints

$\tau_0 = \tau_1$

τ_6 = lookup(x, insert(x, τ_4, {}))

>=(τ_5, τ_6)

τ_2 = gen({}, $\tau_4 \to \tau_5$)

τ_9 = lookup(id, insert(id, τ_2, {}))

>=(int $\to \tau_1$, τ_9)

Current constraint: $\tau_0 = \tau_1$
Result: Success
$\tau_0 = \tau_1$
Substitutions:
$\tau_3 = \tau_4 \to \tau_5$
τ_7 = int $\to \tau_1$
τ_8 = int

Fig. 11. Solver View

The *Type Derivation View* generates the internal derivation tree for an expression entered at the bottom text field. Let us consider for example the expression "`let id = \x.x in id 3`", whose derivation tree is shown in Fig. 10.
The derivation tree of our example expression is displayed right above the text field. On the left of the tree the global information for the type checking process are displayed, at the moment this is only the global context which is passed on to the constraint solvers. The derivation tree contains only judgements, no constraints. To view the constraints the user can hover with the mouse over the rule name, to get a tooltip with the constraints or he can click on the rule name. Clicking on the rule name will fill the right side with information about that rule instance, namely the local context and the constraints arising from this rule.
The last view is the *Solver View*. In this view the user can step through the constraint solving process. The left side lists the constraints yet to be solved. The right side displays the current constraint considered by the solver, the solution of that constraint and the accumulated result substitution. At the bottom right side are the two buttons to step through the solving process.

To run this gui tool, the user needs to provide three things: a list of named inference rules, an initial global context and a function to parse expressions. For our MiniML example the tool could be run as following:

```
main = runGui GuiConfig { namedRules    = rules
                        , globalContext = empty
                        , parseString   = parse
                        }

rules :: [(String,Rule)]
rules = [ ("true", true)
        , ("false", false)
        , ("letrec", letrec)
        , ("varpoly", varpoly)
        , (...) ]
```

9 Related Work

9.1 Tools

Starting in the early 1980s, research concentrating on formal descriptions of programming languages with the goals of generating programming environments and reasoning formally about the specification and implementation led to various ways of expressing type checkers in a given formalism.
Teitelbaum and Reps presented the *Synthesizer Generator* [31,26], a system to generate language-specific editors from descriptions of imperative languages with finite, monomorphic type systems like PASCAL, ADA or MODULA. They used attributed grammars to express the context sensitive part of a language's grammar and the generated editors provided knowledge about the static semantics of the language such that immediate feedback on errors could be given to the programmer.

Bahlke and Snelting developed the *Programming System Generator (PSG)* [1], a generator for language-specific programming environments. The generated environments, focussing mainly on interactive and incremental static analysis of incomplete program fragments, consisted of a language-based editor, an interpreter, and a fragment library system. Using context relations, *PSG* employed a unification-based algorithm for incremental semantic analysis to be able to immediately detect semantic errors even in incomplete program fragments.

Borras et al. introduced the logic engine *Typol* as part of the *Centaur* system [2]. The user of the system was able to state the static and dynamic semantics of a language with the help of inference rules in TYPOL. Those specifications are then compiled to PROLOG for execution.

A system with similar aims is Pettersson's *Relational Meta Language (RML)* [24]. *RML* is a statically strongly typed programming language intended for the implementation of natural semantics specifications. The basic procedural elements are relations: many-to-many mappings defined by a number of axioms or inference rules. Pettersson presents a compiler based on translating *RML* to Continuation-Passing-Style which generates code that is several orders of magnitude faster than *Typol*. *RML* was used for developing of a formal specification of the modeling language Modelica [15].

One of the starting points for the research presented in this paper was Gast's *Type Checker Generator (TCG)* [6], a system which focusses on the generation of type checking functionality exclusively. Gast presents an abstraction for type systems based on logical systems and proposes *type-checking-as-proof-search* as a suitable technique for the implementation of a type checker generator. In this approach type systems can be understood and formalized as logical systems such that there is a typing derivation if and only if there is a proof in the logical system. Unfortunately this approach has one major disadvantage: deduction rules need to be re-factored to make them suitable for proof search, i.e., the formulation of a type system needs to be tailored very precisely to the used technique of the type checker generator. This overhead is acceptable and quite wanted given the design of the *TCG* tool. The generator tool does not work in a standard compiler compiler way and instead of supplying a source file containing the type checker, the user interface to the generated type checking functionality is actually a graphical one. This so called "inspector" layer allows the designer of a type system to trace the type check procedure in a fine-grained manner and brings *TCG*'s intended use to light: it supports the user during the design of a type system by providing an ad-hoc, completely traceable type checker prototype.

While *TCG* focusses on the generation of type checkers, Dijkstra and Swierstra's *Ruler* system [5] tries to bridge the gap between the formal description and the implementation of a type checker. They present a domain specific language for describing typing rules and their system is able to generate an attribute grammar based implementation as well as a visual rendering of the system suitable for the presentation of formal aspects. As part of their work the authors state two problems for which the *Ruler* system provides a sufficient solution:

Problem 1: It is difficult to keep separate formal descriptions and implementations of a modern programming language consistent.

Ruler maintains a single description of the static semantics of a programming language from which material which can be used as a starting point for formal treatment as well as the implementation can be generated.

Problem 2: The Extension of a language with new features means that the interaction between new and old features needs to be examined.

The *Ruler* language allows the user to describe type rules incrementally and makes it easy to describe language features in relative isolation. Separate descriptions of language features can be combined into a description of the complete language and the system is able to check the well-formedness of such a *Ruler* program.

In contrast to the generator approach used by the tools mentioned so far, Kollmansberger and Erwig developed the library *Haskell Rules* [17], a domain-specific embedded language that allows semantic rules to be expressed as HASKELL functions. The library captures many of the thoughts presented in this report, like meta variables in rules (which are called logical variables), unification, substitution, and the lifting and delaying of functions operating on logical values. Judgements formalize the relationship of input and output types in rules, which allows a set of rules to be associated with a typed relationship. Inference in their system is done by a non-deterministic backtracking monad which handles unification, application of substitutions, delaying of functions over logical values, and the generation of fresh logical variables.

Our work takes on several ideas from both *TCG* and the *Ruler* system but enhances them at several points. We chose a well know extension of the standard inference rule based way to formalize type systems and presented a framework which allows us to derive type checking functionality from such type systems. This allows the user to formalize her type system in a standard way and requires minimum overhead when encoding the type rules to be used with our library. From a software engineering point of view we liked *Ruler*'s idea to provide a specific language to define typing rules but we were aiming at a closer integration of our system into the host language so we rejected the generator approach used in *TCG* and *Ruler* in favor of deploying our system as a library such as *Haskell Rules*. Thus our library provides a domain-specific embedded language [13] for defining constraint-based inference rules and a framework for interpreting such rules.

9.2 Constraint-based Typing

Glynn, Sulzmann, and Stuckey presented a general framework for type class systems based on *Constraint Handling Rules* (CHRs) [7]. Constraint handling rules are a multi-headed concurrent constraint language for writing incremental constraint solvers. CHRs define transitions from one constraint to an equivalent constraint and those transitions are used to simplify constraints and detect satisfiability and unsatisfiability. As part of the author's work CHRs are used to define constraint relations among types as an extension to Herbrand (equational) constraints and the CHR constraint solving process is an extension of Herbrand constraint solving to arbitrary newly defined constraint relations. The authors introduce decidable operational checks which enable type inference and ambiguity checking. By type inference for type classes they consider the individual tasks of ensuring that class and instance declarations are compatible, constraints arising in programs are satisfiable, and type schemes provided by the user are valid.

Sulzmann, Odersky, and Wehr introduced a general framework HM(X) for Hindley/Milner style type systems with constraints [23,30]. Particular type systems can be obtained by instantiating the parameter X to a specific constraint system. The Hindley/Milner system itself is obtained by instantiating X to the trivial constraint system over a one point domain.
Following the lead of constraint programming they treat a constraint system as a cylindric algebra with a projection operator. This allows them to formulate a logically pleasing and pragmatically useful rule for quantifier introduction. Additionally, projection is an important source of opportunities for simplifying constraints. As part of the HM(X) framework, simplifying means changing the syntactic representation of a constraint without changing its denotation.

Two of the main strengths of the Hindley/Milner system are the existence of a principal types theorem and a type inference algorithm. The authors state sufficient conditions on the constraint domain X so that the principal types property carries over to HM(X). Based on these conditions they present a generic type inference algorithm that will always yield the principal type of a term and discuss some typing features like extensible records, type classes, overloading, and subtyping as part of their framework.

As part of the Helium project Heeren, Hage, and Swierstra used constraint-based type inferencing in a compiler covering a significant subset of the Haskell language [11]. The major motivation of this project was to yield understandable and appropriate type error messages for novice functional programmers. Instead of using a single deterministic type inference process which works best in all cases for everybody, they advocate the use of a more flexible system that can be tuned by the programmer. To obtain the desired flexibility the authors chose a constraint-based approach and divided the type inference process into three distinct phases:

1. the generation of constraints in the abstract syntax tree,
2. the ordering of constraints in the tree into a list, and
3. the solving of constraints.

This separation has resulted in the Top framework, a Haskell library for building program analysis that offer high quality feedback to users [10]. The generality of their framework allows the simulation of well-known type inference algorithms such as $\mathcal{W}$ [4] and $\mathcal{M}$ [19], by choosing an appropriate order in which the type constraints should be solved.

10 Conclusion and Future Work

We presented the design and implementation of a library capable of deriving type check functionality from a type system's formal description. We proposed constraint-based inference rules to form a suitable formalism to accomplish this task and implemented a library which works in the fashion of an interpreter for typing rules. The initial feature set of our library was determined by two languages chosen as case studies. Their type systems could be adapted to our constraint-based setting in a straightforward manner and we were able to derive type checkers for both languages.
Nevertheless, the library is still more a proof-of-concept implementation than a production quality tool. We tried to take many "real world" requirements such as error messages and the use of auxiliary functions in deduction rules into account but the implementation still carries some non-negligible limitations.
One major drawback of the current version of the framework is the fixed type representation. As with the abstract syntax for expression, the abstract syntax for types is typically custom-tailored for the language to be implemented and a production quality tool should provide certain means to allow the user to define

her own type representation. This shortcoming is merrily do to technical reasons, namely the use of existential types for judgements and meta-level functions prevents us from making the type class `Var` polymorphic in its used data structure for types.

Additionally, there still is certain overhead for the user to make her abstract syntax ready to be used within our type checking framework. To some extend we managed to prevent the user from writing boilerplate code when it comes to defining instance declarations for some type classes by providing *Template Haskell* functions accomplishing this task. But even so we provide this functionality to the user one flaw still remains: these parts of the "internal machinery" are visible to her and from a software engineering point of view this should be considered bad style. This limitation could be overcome easily by using data type generic programming to accomplish tasks like instantiating meta-level elements, first order unification, or collecting type variables contained in certain elements. Unfortunately there is no data type generic programming library available in the HASKELL universe which provides means to handle existential types without requiring any additional information by the user.

To be able to turn the current prototype implementation into a production quality tool, these two crucial limitations need to be taken care of. Since these flaws could not be overcome in the current host language of the project, the only way to handle them would be to port our library to a language providing mechanism or language features to handle these drawbacks accordingly. For a number of reasons, the functional, object-oriented multi paradigm language SCALA [22] seems to be a promising language to implement the library described in this report without the limitations stated earlier. First of all, *subtyping* is a great language feature for writing frameworks, especially when it comes to defining algorithm templates where the user can hook her own data structures in. Additionally, the SCALA language processing library *Kiama* [16] ships with a term rewriting library which provides functionality similar to the ones found in common data type generic programming libraries. And since SCALA 2.10 will introduce a macro/template system, algorithmic program construction for the generation of boilerplate code is still possible. Furthermore SCALA's notion of *implicit conversions* will allow a more convenient use of auxiliary functions in deduction rules since the lifting of such a function call into the corresponding wrapper data structure can be done "under the hood". Last but not least, SCALA's syntactical features make it a great host language for embedding domain specific languages.

Besides the mentioned modifications on the current implementation, future work should include several enhancements providing the user of the framework more flexibility when developing type checkers with the help of our library.

On the one hand it might be discussed whether the *first-fit-rule-matching semantic* of the constraint generation function limits the expressiveness of our library in a notable form. When instantiating the first matching rule during the constraint generation phase our library formulates one distinct requirement to the type rules in order to be able to derive type checking functionality from them: the rules must be syntax-directed, i.e., the conclusions of the formulated rules cannot be overlapping.[4] Thus a type system needs to be re-factored in two ways in order to derive a type checker from it with the help of our library: its type rules need to be formulated constraint-based and syntax-directed. To handle rules with overlapping conclusions the techniques used by the library need to be adjusted slightly. At the moment, one constraint set is generated for a given program and the program is well typed, if and only if these constraints have a solution. Allowing declarative typing rules, the constraint generation phase needs to mimic the backtracking in the type derivation by computing a set of constraint sets for a given program and the program is well typed, if and only if at least one constraint set has a solution which is equal to the solutions of all other solvable constraint sets.

Additionally, the framework could be enhanced at several points providing more flexibility for the user when defining type checkers. Following the ideas presented in the *TOP* framework [10], our constraint-based setting should be adapted such that three phases are used instead of just two:

1. Collecting Constraints
2. Ordering Constraints
3. Solving Constraints

In this setting, the constraint generation phase would yield a tree labeled with constraints. This tree is flattened in the next phase using tree walks or tree transformers. The library should provide a default set of flattening strategies (like top-down or bottom-up), but the user should still be able to define her own ordering strategies. Last but not least the constraint solving phase should be improved by allowing the user to employ different solving techniques, e.g., a greedy solver or even more sophisticated approaches like type graphs for global analysis [12,9].

In addition with some more infrastructure for error messages (positions, etc.) these improvements and modifications should push the implementation towards a level of expressiveness and flexibility which should enable us to define production quality type checkers with the help of our library.

[4] Pierce calls such rules algorithmic and discusses this matter in the context of subtyping [25].

References

1. Bahlke, R., Snelting, G.: The PSG System: From Formal Language Definitions to Interactive Programming Environments. ACM Transactions on Programming Languages and Systems 8(4), 547–576 (Oct 1986)
2. Borras, P., Clement, D., Despeyroux, T.: Centaur: The System. In: 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 3). pp. 14–24. ACM New York (1988)
3. Clément, D., Despeyroux, T., Kahn, G., Despeyroux, J.: A Simple Applicative Language: Mini-ML. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 13–27. New York, USA (Aug 1986)
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 207–212 (Jan 1982)
5. Dijkstra, A., Swierstra, S.D.: Ruler: Programming type rules. In: Functional and Logic Programming: 8th International Symposium (FLOPS 2006). vol. 3945/2006, pp. 30–46. Springer-Verlag, Fuji-Susono, Japan (Apr 2006)
6. Gast, H.: A Generator for Type Checkers. Phd thesis, Eberhard-Karls-Universität Tübingen (2004)
7. Glynn, K., Stuckey, P.J., Sulzmann, M.: A General Type Class Framework. Tech. rep., Dept. of Computer Science and Software Engineering, Univerity of Melbourne (2001)
8. GTK+: http://www.gtk.org/
9. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In: 18th International Symposium on Implementation and Application of Functional Languages (IFL '06). pp. 199–216. Springer Berlin/Heidelberg, Budapest, Hungary (2005)
10. Heeren, B.: Top Quality Type Error Messages. Phd thesis, Universiteit Utrecht (Sep 2005)
11. Heeren, B., Hage, J., Swierstra, S.D.: Constraint Based Type Inferencing in Helium. In: Workshop Proceedings of Immediate Applications of Constraint Programming. pp. 59–80. Cork, Ireland (2003)
12. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the Type Inference Process. In: 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03). pp. 3–13. ACM New York, Uppsala, Sweden (Sep 2003)
13. Hudak, P.: Modular Domain Specific Languages and Tools. In: Proceedings of the Fifth International Conference on Software Reuse. pp. 134–142. Victoria, BC , Canada (1998)
14. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (May 2001)
15. Kagedal, D., Fritzson, P.: Generating a Modelica compiler from natural semantics specifications. In: Summer Computer Simulation Conference (SCSC '98). No. Modelica, Reno, Nevada, USA (1998)
16. Kiama: http://code.google.com/p/kiama/
17. Kollmansberger, S., Erwig, M.: Haskell Rules: Embedding Rule Systems in Haskell. Draft Paper (Jun 2006)
18. Lämmel, R., Peyton Jones, S.: Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03). vol. 38, pp. 26–37 (2003)

19. Lee, O., Yi, K.: Proofs about a Folklore Let-Polymorphic Type Inference Algorithm. ACM Transactions on Programming Languages and Systems 20(4), 707–723 (Jul 1998)
20. Levin, M.Y., Pierce, B.C.: TinkerType: A Language for Playing with Formal Systems. Journal of Functional Programming 13(02), 295–316 (Mar 2003)
21. Mitchell, N., Runciman, C.: Uniform Boilerplate and List Processing. In: Proceedings of the ACM SIGPLAN Haskell Workshop. pp. 49–60. Freiburg, Germany (Sep 2007)
22. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press, 1st edn. (2008)
23. Odersky, M., Sulzmann, M., Wehr, M.: Type Inference with constrained Types. Theory and Practice of Object Systems 5(1), 35–55 (1999)
24. Pettersson, M.: RML - A New Language and Implementation for Natural Semantics Mikael. In: 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP '94). pp. 117–131. Springer-Verlag, Madrid, Spain (Sep 1994)
25. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
26. Reps, T., Teitelbaum, T.: The Synthesizer Generator: A System for Constructing Language-Based Editors. Springer-Verlag, New York, NY, USA (1989)
27. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (Jan 1965)
28. Rodriguez Yakushev, A.L.: Towards Getting Generic Programming Ready for Prime Time. Phd thesis, Utrecht University (2009)
29. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices 37(12), 60–75 (Dec 2002)
30. Sulzmann, M.: A General Framework for Hindley-Milner Type Systems with Constraints. Doctoral dissertation, Yale University (2000)
31. Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Communications of the ACM 24(9), 563–573 (Sep 1981)
32. The Type Checker Library: https://projects.uebb.tu-berlin.de/typechecklib/trac/
33. Wand, M.: A Simple Algorithm and Proof for Type Inference. Fundamenta Informaticae 10, 115–122 (1987)
34. wxWidgets: http://www.wxwidgets.org/

Appendix A: Default Abstract Syntax

For rapid prototyping of type checking functionality the libary provides abstract syntax ready to use with our type checking framework. This abstract syntax consists of terms, definitions, and combiners. Terms allow the user to encode variables, constants, characters, binder, and tagged combiner. Definitions bind identifier to terms and container encapsulate sub-containers and definitions. All data structures provide corresponding meta-level elements and allow the user to define sets, sequences, and meta-level functions over these elements.

```
data Tag = Let                   -- Let binding
         | LetRec                -- Recursive let binding
         | App                   -- Application
         | Abs                   -- Lambda abstraction
         | IfThenElse            -- Conditional
         | Fix                   -- Fixpoint combinator
         | Tuple                 -- n-ary tuple
         | Tag String            -- 'String'-based tag

data Term = Nil                     -- Empty term
          | Var    Ide              -- Variable
          | Const  Integer          -- Constant
          | Char   Char             -- Character
          | Bind   Ide Term         -- Object level binder
          | K      Tag Int [Term]   -- Tagged combiner
          | TSeq   (Sequence Term)  -- Sequence of terms
          | TSet   (Set Term)       -- Set of terms
          | TFun   (MetaFun Term)   -- Embedded function
          | MConst String           -- Meta-level constant
          | MChar  String           -- Meta-level character
          | MTerm  String           -- Meta-level term

data Def = Def   Ide Term        -- Definition
         | DSeq  (Sequence Def)  -- Sequence of definitions
         | DSet  (Set Def)       -- Set of definitions
         | DFun  (MetaFun Def)   -- Meta-level function
         | MDef  String          -- Meta-level definition

data C = C    [C] [Def]          -- Container
       | CSeq (Sequence C)       -- Sequence over container
       | CSet (Set C)            -- Set over container
       | MC   String             -- Meta-level container
```