Safely Speaking in Tongues

Statically Checking Domain Specific Languages in Haskell

Matthew Sackman and Susan Eisenbach

Imperial College, London,
{ms, sue}@doc.ic.ac.uk

Abstract. Haskell makes it very easy to build and use Domain Specific Languages (DSLs). However, it is frequently the case that a DSL has invariants that can not be easily enforced statically, resulting in runtime checks. This is a great pity given Haskell's rich and powerful type system and leads to all the usual problems of dynamic checking.
We believe that Domain Specific Languages are becoming more popular: the internet itself is a good example of many DSLs (HTML, CSS, JavaScript, Flash, etc), and more seem to be being added every day; most graphics cards already accept programs written in the DSL OpenGL Shading Language (GLSL); and the predicted growth of heterogeneous CPUs (for example IBM's Cell CPU) will demand many different DSLs for the various programming models and instruction sets that become available.
We present a technique that allows invariants of any given DSL to be lifted into the Haskell type system. This removes the need for runtime checks of the DSL and prevents programs that violate the invariants of the DSL from ever being compiled or executed. As a result we avoid the pitfalls of dynamic checking and return the user of the DSL to the safety and tranquillity of the strongly statically typed Haskell world.

1 Introduction

Algebraic Data Types (ADTs) provide a very easy way to design and build a Domain Specific Language (DSL) with some degree of type safety. For example, a simple expression language can be created very easily:

data *Expr* = *Bool Bool* | *Int Int*
| *Plus Expr Expr* | *Times Expr Expr* | *Eq Expr Expr*

However, this does not prevent a *Bool* being compared for equality (*Eq*) with an *Int*, which we might well want to reject rather than just return false. Even worse, we can attempt to multiply an *Int* by a *Bool*, which we definitely want to reject. Here the problem is that we have two distinct types, integers and booleans, and we must limit the constructors to only accepting expressions of the correct type. Thus we add a type parameter to track the type of the current expression and we switch to using Generalised Algebraic Data Types (GADTs) [1]. GADTs allow the precise type signature of a constructor to be specified explicitly. Using a GADT rather than a number of plain ADTs offers the simplest solution as we don't need to build a hierarchy of data types:

```
data Expr t where
  Bool  :: Bool → Expr Bool
  Int   :: Int → Expr Int
  Plus  :: Expr Int → Expr Int → Expr Int
  Times :: Expr Int → Expr Int → Expr Int
  Eq    :: Expr t → Expr t → Expr Bool
```

Eq is particularly satisfying here as it allows any two expressions of the same type to be compared for equality, thus there is no need for separate *Eq* constructors for boolean and integer expressions.

If the DSL is to contain statements rather than expressions, and has state, then things can become rather more tricky. Making further use of GADTs to model variables and variable creation, and combing these GADTs with the *State* monad certainly allows you to statically enforce some invariants. But not all desired invariants can be enforced with these techniques. For example, consider a DSL for the assembly language with CPU-registers. The value within a register has a type, and you may wish to ensure that you can only do sensible type-safe things with that register. But the register could load another value with another type, thus the type of the register has to be able to vary. The GADT approach cannot automatically consider that the register can change type and so would require explicit constructs in the DSL to permit this. Another example is the security analysis (the Bell / LaPadula system) that we present in section 5: this cannot be implemented through GADTs alone.

It is very common then for GADTs and related techniques to be used to enforce as much as possible, but then for certain checks to be performed at runtime. This often provides a false sense of security to the user of the DSL as they may not realise that their DSL-program may be rejected at runtime, and it also complicates matters significantly for the library writer and DSL-designer as they must ensure that they check the DSL-program at every necessary point in the provided API. Forgetting to check the DSL in just one public function may be enough to expose privileged information or to at least bring down a system. Further, if the same DSL-program is used in multiple places, the checking would have to be done multiple times. This could become very expensive if the DSL-program is large and the checking complex. It should not be difficult to see the benefits of being able to statically check the DSL-program.

- We present a technique that allows such complex invariants to be statically checked by lifting the invariants into the Haskell type system.
- We demonstrate our technique with a running example using an extended form of the While language as our DSL (section 2) which permits both boolean and numeric variables and for variables to change their type. After explaining our framework (section 3) we use it to implement type inference (section 4) and then extend the While language with a security model (Bell / LaPadula) which gives secrecy levels to variables and enforces restrictions on information flow (section 5).

DSLs are not a new technique and have been successfully used for a variety of applications. SQL is a very widely used DSL for accessing databases and has

```
Bool ::= True | False
Num ::= 0, 1, 2, ... ∈ ℕ
Var ::= a, b, c, ... ∈ 𝕍
Expr ::= BoolExpr | NumExpr
Statement ::= skip
            | if BoolExpr
                 then Statements
                 else Statements
            | while BoolExpr
                 Statements
            | Var := Expr
Statements ::= ϵ
             | Statement
               Statements

BoolExpr ::= Bool
           | Var
           | not BoolExpr
           | or BoolExpr BoolExpr
           | and BoolExpr BoolExpr
           | NumExpr < NumExpr
           | NumExpr > NumExpr
           | Expr == Expr
NumExpr ::= Num
          | Var
          | negate NumExpr
          | NumExpr + NumExpr
          | NumExpr - NumExpr
          | NumExpr * NumExpr
          | NumExpr / NumExpr
```

Fig. 1: The syntax of the While language

been subjected to many techniques to statically enforce its invariants. We discuss these and other related work in section 6, before concluding in section 7.

2 The While Language

We demonstrate our technique by enforcing invariants on a simple While language. The While language is a very basic but general purpose imperative language, that is statement based using variable assignment (`:=`), has typical arithmetic, boolean and logical functions, and has `while` and `if` control flow constructs. The `if` must have both branches specified and there is a no-op instruction, `skip` which is typically used in a branch of an `if` to effectively eliminate that branch. Using a general purpose language (albeit it a very simple one) as our DSL demonstrates how general our technique is, and offers evidence that our techniques can be adapted and used in far more specific DSLs.

The original While language [2] only permitted variables to be assigned numeric expressions and boolean expressions only appeared in the conditions of `while` and `if` statements. We extend the While language so that boolean expressions can be assigned to variables, and that variables can change their type, provided it is done in a type safe way. Variables can be created inside blocks and are lexically scoped.

The syntax of the While language is shown in figure 1. Here the definition uses separate types for boolean and numeric expressions. This allows the definitions of monomorphic functions to be specified correctly (e.g. it is not possible to

multiply a number by a boolean), but equality (`==`), which is polymorphic, is poorly specified. Typically at this point, the operational semantics would only define reduction where equality is used in a type-safe way, but we wish to be able to statically reject such incorrect use of equality as an invariant of the language rather than as a consequence of the operational semantics. Furthermore, we see that a Var can be both a numeric or a boolean value and can change type. We want the type system to reject programs that do unsafe things with variables.

One of the invariants we wish to enforce is that a variable can never be of an ambiguous type: that is, a variable can not change type within the body of a `while` loop (as the `while` loop may never be entered) and if it changes type within an `if` statement then both branches must result in the variable being the same type. This is a slightly stronger property than is absolutely necessary: it could be legal to change the type of variable within the body of a `while` loop or for an `if` statement to leave a variable with two types (or more, given nestings), provided that the variable is subsequently only assigned to and not read from. This would require a *def-use* analysis which is possible to implement with our technique, however, here we enforce only the simpler property.

With GADTs alone, you could implement many of these invariants statically. However, frequently the DSL would have to be extended to support additional operations directly, for example adding explicit constructs to change the type of a variable, which somewhat goes against the notion of type *inference*, and additionally makes the DSL larger and more complex. In our opinion, much of the value of a DSL is in its ease of use and forcing the user to compromise on this front to facilitate analyses is something to be avoided.

3 Static Analysis Framework

Our aim is to treat the type system as a programming environment as this will allow us to statically enforce the invariants as required. We shall rely on a library of functions that we have built, providing type level numbers, lists (which we frequently treat as sets) and maps and cover their APIs as necessary. Wherever possible we have tried to copy the APIs from normal value-level libraries such as Haskell's *Data.List* library in order to make these libraries as predictable and familiar as possible. In many cases, the only difference in use between our type-level versions and the normal value-level versions are that in using the type-level version you must put a type class constraint in the context of your function.

When checking a DSL-program, we need to be able to walk over the structure of the program, collecting information and checking that invariants are satisfied by each statement. To do this we ensure that the type of one statement depends upon the type of the preceding statement, thus analysis of the preceding statement must occur before the current statement can be typed. Hence we achieve type-level computation for each statement. These dependencies are created by using a type indexed monad:[1]

[1] In this presentation we redefine the standard Haskell *Monad* type class as shown, which is legal Haskell 98 but isn't accepted by GHC prior to version 6.10. For versions

```
class Monad m where
  (>>)   :: m x y a → m y z b → m x z b
  (>>=)  :: m x y a → (a → m y z b) → m x z b
  return :: a → m x x a
```

Thus the monad has gained two extra type parameters, the first can be thought of as the *from* state, and the second can be thought of as the *to* state. Now we can see that if we have statements of the While-DSL which are combined together using these monadic functions, then the type of the second statement must have a *from* state equal to the *to* state of the first statement, thus the dependency is achieved. It is into these type parameters that we shall place all the state that we require to check the properties of our DSL.

This monad can very easily be thought of as nothing more than the normal *State* monad except that the type of the state can change. The only instance of this monad that we need is similarly very close to the definition of the standard *State* data type in Haskell:

```
newtype State x y a = State{ runState :: x → (a, y)}

instance Monad State where
  f >> g   = State (λx → let (_, y) = runState f x in runState g y)
  f >>= g  = State (λx → let (a, y) = runState f x in runState (g a) y)
  return a = State (λx → (a, x))
```

So, in this framework, all we need is to define a data structure that will form the *from* and *to* states (i.e. the x and y parameters to the *State* type) and to define the functions that will allow the user to write a DSL-program.

For the While-DSL, we need to be able to identify variables uniquely at the type level in order to track their type and check their use. This requires that each new variable has a different type, and so we use type-level numbers to identify each variable. Thus the state must contain the next number to be used for the next variable. This shall be incremented as variables are created. Set membership is used for tracking the type of each variable. We need three sets to hold integer variables, boolean variables and newly created variables which have not yet been assigned to and thus have no type. The functions that we define which make up the While language will manipulate these sets and types in order to enforce the invariants as required.

Our framework can produce a value if the invariants of the DSL are satisfied, and for the While-DSL, we choose to produce an Abstract Syntax Tree (AST) of the program that has been supplied. This output is determined by the DSL-designer, who could equally decide no output is required and just use the framework for enforcing the DSL's invariants. We use a very simple ADT to produce the AST, one which does not enforce the invariants we are interested in. However, this ADT is never exposed to the user so is considered safe to use internally within the library. In this way, our framework acts as the bridge between the unsafe external world, statically rejecting bad DSL-programs, and the safe, controlled inner world of the library or DSL-designer. The ADT is shown in

---

of GHC prior to 6.10, we have to rename the monad functions and abandon **do**-syntax. Using **do**-syntax makes the presentation more familiar and simpler.

```
type Statements
  = [Statement]
data Statement
  = Skip
  | Var := Expr
  | If BoolExpr Statements Statements
  | While BoolExpr Statements
    deriving (Show, Eq)
newtype Var
  = Var String
    deriving (Show, Eq)
data Expr
  = Boolean BoolExpr
  | Integer IntExpr
    deriving (Show, Eq)
```

```
data BoolExpr
  = BoolTerm Bool
  | BoolVar Var
  | Not BoolExpr
  | Or BoolExpr BoolExpr
  | And BoolExpr BoolExpr
  | Eq Expr Expr
  | LT IntExpr IntExpr
  | GT IntExpr IntExpr
    deriving (Show, Eq)
data IntExpr
  = IntTerm Int
  | IntVar Var
  | Negate IntExpr
  | IntExpr :+: IntExpr
  | IntExpr :*: IntExpr
  | IntExpr :/: IntExpr
  | IntExpr :-: IntExpr
    deriving (Show, Eq)
```

Fig. 2: The Algebraic Data Type of the While-DSL

figure 2 and is extremely similar to the syntax of the While language in figure 1. From this ADT, it is straightforward to write an interpreter for the While-DSL which walks over the AST, if that is what the DSL-designer desires.

To create the AST, we need to be able to convert variables that are identified by types into Strings, i.e. when a variable is created, it will have a unique identifying type (a type-level number), e.g. *D5 E*, but it will also have a name which is a String, for example `"f"`. Our type-level numbers support converting a type-level number to an *Int*, so we have a *Map* from *Int*s to *String*s which will allow us to convert a variable as a type to its name for use in the AST, a list of *String*s to act as a source of value-level names for the variables, and a list of *Statement*s which will form the resulting AST of the While-program. So, as the While-DSL designer, we define the state as:

```
data DSLState nextVar unusedVars boolVars intVars
  = DS{ names      :: [String],
        nameMap    :: Map Int String,
        statements :: AST.Statements,
        nextVar    :: nextVar,
        unusedVars :: unusedVars,
        boolVars   :: boolVars,
        intVars    :: intVars }
```

and our functions will therefore be variations upon the type:

```
State (DSLState nextVar  unusedVars  boolVars  intVars)
      (DSLState nextVar' unusedVars' boolVars' intVars') result
```

permitting the types that hold the state we are interested in to vary as necessary. It is the relationships between these type parameters, enforced through type class contexts, which will allow the calculation of the *to* state from the *from* state, and will either accept or reject the provided While-program.

To run the framework, we need an initial state from which analysis will begin and a function that will supplies the initial state to the analysis. Considering the purpose of each component of the state, the initial state is clear: the *nextVar* will be a type level number of zero (represented by *D0 E*), and the *unusedVars*, *boolVars* and *intVars* are all the empty list (*Nil*). We define the function *buildWithInvariants*, which applies this initial state to the While program supplied, and produces the AST as the output:

```
buildWithInvariants :: State (DSLState (D0 E) Nil Nil Nil)
                             (DSLState nextVar Nil boolVars intVars) () →
                        AST.Statements
buildWithInvariants prog = statements (snd (runState prog initialState))
  where
    initialState = DS{ names      = names,
                       nameMap    = Map.empty,
                       statements = [],
                       nextVar    = (D0 E),
                       unusedVars = nil,
                       boolVars   = nil,
                       intVars    = nil}
    names = [[x] | x ← ['a'..'z']] ++
            [reverse (x : y) | y ← names, x ← ['a'..'z']]
```

4 Type Inference

As the While-DSL designer and invariant enforcer, there are just five functions we need to implement: the four statements of the While-DSL and the ability to create new variables. The easiest of these is the `skip` statement which has no impact on typing whatsoever as it is a no-op. The implementation is very simple:

```
skip :: State (DSLState nextVar unusedVars boolVars intVars)
              (DSLState nextVar unusedVars boolVars intVars) ()
skip = State f
  where
    f ds@(DS{ statements})
      = ((), ds{ statements = statements ++ [AST.Skip]})
```

Next we tackle creating a new variable. We need to increment the type *nextVar*, return and insert the old value into the *unusedVars* set, and create a corresponding value-level representation of the variable, as a *String*, shown in figure 3. *Num* is our module for type-level numbers and we see that to find the successor of a number is nothing more than putting the *Succ* type class in the context of our function. Similarly, with the *NumberToInt* type class (and corresponding *numberToInt* function), which converts a type-level number to an

```
newVar :: ∀ nextVar nextVar' unusedVars unusedVars' boolVars intVars .
        (Num.Succ nextVar nextVar',
         SetCons nextVar unusedVars unusedVars',
         NumberToInt nextVar) ⇒
  State (DSLState nextVar  unusedVars  boolVars intVars)
        (DSLState nextVar' unusedVars' boolVars intVars) nextVar
newVar = State f
  where
    f ::  (DSLState nextVar unusedVars boolVars intVars) →
          (nextVar, (DSLState nextVar' unusedVars' boolVars intVars))
    f ds@(DS{ names = (name : names), nameMap, nextVar, unusedVars})
      = (nextVar, ds{ names = names, nameMap = nameMap',
                      nextVar = nextVar', unusedVars = unusedVars' })
        where
          nameMap' = Map.insert varNum name nameMap
          nextVar' = Num.succ nextVar
          varNum = numberToInt nextVar
          unusedVars' = setCons nextVar unusedVars
```

Fig. 3: Creating a new variable

Int, and the *SetCons* type class which operates on type-level lists, performing a *cons* as normal except when the element is already a member of the list, thus a list constructed in this way will not contain duplicates.

The three remaining functions (`:=`, `while` and `if`) all involve expressions so it is these we must tackle next. When a variable is used in an expression we do not know whether its current value is a boolean or integer, but the expression in which it is used demands that the variable's value is a particular type. Thus we need to check that the variable's current type, held in the type-level sets in the state, matches the use of the variable in the expression. A straightforward expression ADT would simply give us a value of type *Expression* which would be useless to us as a type, as we need a much richer type to be able to verify that variables are used correctly. Thus we use a GADT to build an expression such that the type of the expression contains the variables and the way in which they are used. Our idea is that the expression GADT has a type parameter which represents a stack, such that every element of the stack is a tuple, where the left of the tuple is the desired type, and the right of the tuple is a list of variables which are required to have the type on the left. For example:

$$Cons\ (Int, (Cons\ X\ (Cons\ Y\ Nil)))\ (Cons\ (Bool, Cons\ Z\ Nil)\ Nil)$$

is a type-level stack in which X and Y are variables which we need to be *Int*s and Z is a variable we need to be a *Bool*.[2] The expression GADT has three type parameters, the last of which is this stack structure, and the other two are nothing more than the head of the stack, decomposed, which simplifies construction of the stack. The *Expr* GADT is shown in figure 4.[3]

[2] *Cons* is the type-level equivalent of (:), just as *Nil* is the type-level equivalent of [].
[3] *Append* is a type-level version of (++).

```
data Expr ty vars stack where
  BC     :: Bool → Expr Bool Nil Nil
  IC     :: Int → Expr Int Nil Nil
  Var    :: (NumberToInt v) ⇒ v → Expr ty (Cons v Nil) Nil
  Not    :: Expr Bool vars stack → Expr Bool Nil (Cons (Bool, vars) stack)
  And    :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Bool vars stack → Expr Bool vars' stack' →
            Expr Bool Nil (Cons (Bool, vars'') stack'')
  Or     :: ...   -- as for And
  Eq     :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr ty vars stack → Expr ty vars' stack' →
            Expr Bool Nil (Cons (ty, vars'') stack'')
  LT     :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Int vars stack → Expr Int vars' stack' →
            Expr Bool Nil (Cons (Int, vars'') stack'')
  GT     :: ...   -- as for LT
  Negate :: Expr Int vars stack → Expr Int Nil (Cons (Int, vars) stack)
  Plus   :: (Append vars vars' vars'', Append stack stack' stack'') ⇒
            Expr Int vars stack → Expr Int vars' stack' →
            Expr Int Nil (Cons (Int, vars'') stack'')
  Minus, Times, Divide :: ...   -- as for Plus
```

Fig. 4: The *Expr* GADT

Thus we see the constructors for constants (*BC* and *IC*), being leaves of the expression tree, have an empty stack and do not use variables. Using a variable is achieved with the *Var* constructor, which is also a leaf of the expression tree so we construct just a singleton list of the variable, but we don't know at this point the type which the variable is expected to have, thus we leave the type floating as *ty*. The *And* constructor takes two sub-expressions which must both produce *Bool*s, we combine both the subexpressions' lists of variables, and their stacks. Thus in the expression *And* (*Var a*) (*Not* (*Var b*)) the variable *a* would be in the *vars* type parameter for the left child of *And*, so would be placed on the top of the stack by the *And* constructor, whereas the variable *b* would already be in the stack, having been placed there by the *Not* constructor. Thus this GADT only applies types to variables which are direct children of any given constructor, which is what we, as the DSL-designer, require. *Eq* is polymorphic, merely requiring the subexpressions are the same type. The result of equality is a *Bool*; and when building the stack, we use the common *ty* type variable of the subexpressions. This means that the expression *Eq* (*Var a*) (*Var b*) does not have a ground type as the type of the variables *a* and *b* are not fully known: they could both be *Bool*s or *Int*s.

Type checking is then quite simple as we only need to walk over the stack and check that the variables used are members of the correct sets of variables:

```
class ValidVarUse stack boolVars intVars
instance ValidVarUse Nil boolVars intVars
```

instance ($ValidVarUse\ stack\ boolVars\ intVars$,
$IsSubList\ (Cons\ v\ vars)\ intVars\ isInt$,
$IsSubList\ (Cons\ v\ vars)\ boolVars\ isBool$,
$Bool.Not\ isInt\ isBool, Bool.If\ isBool\ Bool\ Int\ ty) \Rightarrow$
$ValidVarUse\ (Cons\ (ty, (Cons\ v\ vars))\ stack)\ boolVars\ intVars$

instance ($ValidVarUse\ varStack\ boolVars\ intVars) \Rightarrow$
$ValidVarUse\ (Cons\ (ty, Nil)\ varStack)\ boolVars\ intVars$

We recurse over the stack, and for each element we have two cases, for when variables have and have not been used. The lack of a ground type for certain expressions means that we can't match *Bool* or *Int* on the type parameter *ty* at the head of the stack as *ty* may be unknown. Instead, we reason that the variables must be a sublist of either the booleans or the integers *but not both* (achieved through the *Not* relation), and then we require that if they are a sublist of the booleans then *ty* must be *Bool* otherwise it must be *Int*; the $Bool.If\ c\ t\ f\ r$ class context should be read as $r =$ **if** c **then** t **else** f. *IsSubList* and the *Bool* module are both provided by our libraries of type-level functions.

For assignment we need one further type class (called *TypeVar*) which updates the sets of variables correctly. The variable being assigned to is removed from its current set and inserted into the set corresponding to the type of the expression. Sadly, this can't be implemented quite this simply as it is a challenge to make GHC understand that removing an element from a set and then adding it straight back in results in the same original set (a property we need for showing that the body of a `while` loop does not alter the type of any variables), thus we have to do some set membership tests first. The type class is otherwise straightforward. Assignment then checks that the expression uses variables correctly (the *ValidVarUse* type class), it updates the type of the variable being assigned to (the *TypeVar* type class), and it suitably adds to the list of *Statement*s for the AST (which requires the *NumberToInt* type class and also the *NumberListToIntList* type class which is a mapping of the former over a list of type-level numbers). The conversion from the *Expr* GADT to the AST is entirely mechanical. The type of assignment is therefore:

$(.=) :: (TypeVar\ var\ ty\ unusedVars\ boolVars\ intVars$
$unusedVars'\ boolVars'\ intVars',$
$ValidVarUse\ (Cons\ (ty, vars)\ stack)\ boolVars\ intVars,$
$NumberToInt\ var,$
$NumberListToIntList\ boolVars, NumberListToIntList\ intVars) \Rightarrow$
$var \rightarrow Expr\ ty\ vars\ stack \rightarrow$
$State\ (DSLState\ nextVar\ unusedVars\ \ boolVars\ \ intVars)$
$(DSLState\ nextVar\ unusedVars'\ boolVars'\ intVars')\ ()$
$var .= expr = ...$

The `if` and `while` statements are very similar so we only present the `while` statement as it is the simpler of the two. The expression is type checked as for assignment. The body may create lexically scoped variables, so variables that are created within the body must not escape into the outer scope. We create a type class *RemoveVars* which takes the range of variables created (i.e.

nextVar to *nextVar′*) and removes variables within that range from the sets of variables. Finally, we require that after removing variables created within the body, the body results in the same sets of variables with the same type as were initially supplied. Thus the body cannot leave a variable in a different type than it was before the `while` statement. The body of the function is not enormously interesting, whereas the type is.

```
while :: (ValidVarUse (Cons (Bool, vars) stack) boolVars intVars,
          RemoveVars nextVar nextVar′ unusedVars′ boolVars′ intVars′
                                      unusedVars  boolVars  intVars,
          NumberListToIntList boolVars, NumberListToIntList intVars) ⇒
  Expr Bool vars stack →
  (State (DSLState nextVar  unusedVars  boolVars  intVars)
         (DSLState nextVar′ unusedVars′ boolVars′ intVars′) ()) →
  (State (DSLState nextVar  unusedVars  boolVars  intVars)
         (DSLState nextVar′ unusedVars  boolVars  intVars)  ())
while cond body = ...
```

In Haskell, **if** is a keyword so cannot be redefined, so we instead call the function if_w. This function uses all the same machinery, just slightly differently. The two bodies of the `if` statement can leave variables with different types than before the `if` statement, provided they both make the same changes.

And that is it: in under 400 lines of Haskell specific to the While-DSL, we can now write While programs, supply them to the *buildWithInvariants* function, and if no error occurs, know that the invariants of the language were statically checked and a simple AST of the program is returned. The code is not particularly complex, and whilst it is more verbose than equivalent runtime checks, the advantages of statically asserting these invariants more than outweighs the extra code size.

5 Extending While with Security

To demonstrate how flexible and extensible this framework is, we extend the While-DSL with the Bell / LaPadula security model [3], which constrains information flow. Variables are given security categories such as *public* or *top-secret*, which are totally-ordered, and the system ensures that no *read-ups* or *write-downs* occur. These can happen in assignments where the expression on the right of the assignment reads a variable which has a higher security level (so a *read-up*) than the variable to which the expression is being assigned (and so represents an information leak) and viewing it the other way around, it is also a *write-down* as privileged information is being written down to a lower level. But it doesn't just occur in assignments: the condition of a `while` or `if` statement imposes a minimum security level on the bodies of those statements for the very fact that a body is being executed reveals information about the conditional. Thus every statement within the body must be at the same security level or higher than is required to read the variables within the conditional expression. This analysis could, for example, be used to ensure that data read from a

database is never directly returned to a user, or that passwords are never held by user-visible variables.

Our implementation extends this further: when a variable is created, we provide two security levels, a minimum and a maximum. These can be the same value, in which case checking is performed as described above. However, the minimum value can be less than the maximum and indicates that we're not sure what the security level of the variable should be. The maximum level will not change, and so if the program is rejected then we know that some variable is not sufficiently privileged given the maximum security levels, but the minimum level can rise, which indicates that the variable must be granted at least the resulting minimum level for the program to be acceptable. The maximum level is used when reading a variable, and the minimum level is used when writing to a variable. Thus raising the minimum security level is always a safe operation, because it only increases what can be written to a variable but does not alter who can read that variable. So raising the minimum level of a variable cannot invalidate assignments or conditionals that have already been checked. Our security levels are type-level numbers, where zero (*D0 E*) is the lowest possible level.

To implement this security analysis, we need to extend the type-level state. We need to carry around a minimum security level: this is set by `if` and `while` conditionals and represents the minimum security level a variable must have to be assigned to within such a statement. We also need a type-level map (provided by our *TMap* module), from variables to tuples representing the current minimum and maximum security level. The `skip` statement has no security implications and so is unaltered. Creating a new variable is as before, except it inserts the minimum-maximum tuple into the security map, and we check the minimum is less than the maximum. Expressions already expose in their type all the variables that are used, so we once again need to be able to walk this structure and extract the highest maximum security level, which we will use as the *read level* of the expression. This is the purpose of the *MaxSecurityLevel* type class:

class *MaxSecurityLevel map varStack maxIn maxOut*
  | *map varStack maxIn* → *maxOut* **where**
  maxSecurityLevel :: *map* → *varStack* → *maxIn* → *maxOut*

The instances of this type class walk first over the stack of tuples and then over the inner lists of variables, indexing the *map* and testing to see if the maximum security level of each variable is greater than the current security level (held by *maxIn*). The maximum level found is returned in *maxOut*.

Assignment then uses the *MaxSecurityLevel*, finding the *read level* of the expression (*exprMax*). We check to see if this is lower than the minimum level currently held by the state (*minExpr*), if so, the state's level is used it its place. This ensures that if the conditional of an `if` or `while` statement uses variables at a particular level (and hence updates the state to hold that level) then assignments within the bodies of those statements must be valid for at least the same security level, if not higher. The variable to which we are assigning must have a minimum level greater than or equal to the *read level*. If this is not the case then we attempt to raise its minimum level sufficiently, but enforce that

this must still remain below its maximum level. Finally we update the security map (*sMap*) with the new security levels. The additional type class contexts are shown below in black and the existing previous code in grey:

```
(.=)::(TypeVar var ty unused bools ints unused' bools' ints',
       ValidVarUse (Cons (ty, vars) stack) bools ints,
       NumberToInt var,
       NumberListToIntList bool, NumberListToIntList ints,
       MaxSecurityLevel sMap (Cons (ty, vars) stack) (D0 E) exprMax,
       IsSmallerThan exprMax minExpr exprMaxSmallerThanMin,
       Bool.If exprMaxSmallerThanMin minExpr exprMax exprMax',
       TMap.Lookup sMap var (varMin, varMax),
       IsSmallerThan varMin exprMax' varMinNeedsRaising,
       Bool.If varMinNeedsRaising exprMax' varMin varMin',
       SmallerThanOrEq varMin' varMax,
       TMap.UpdateVarying sMap var (varMin', varMax) sMap') ⇒
  var → Expr ty vars stack →
  State (DSLState nextVar unused  bools  ints  sMap  minExpr)
        (DSLState nextVar unused' bools' ints' sMap' minExpr) ()
var .= expr = ...
```

The `if` and `while` statements are again very similar. They extract the *read level* of the conditional, *exprMax*, find the maximum of that value and the level currently held by the state (to ensure the minimum level isn't decreased by nested statements), *minExpr'*, and update the state used for their bodies with this value. Thus the final security map, *sMap'*, contains the security requirements of the bodies, which were built using the updated minimum expression level.

```
while::(ValidVarUse (Cons (Bool, vars) stack) bools ints,
        RemoveVars nextVar nextVar' unused' bools' ints'
                                       unused  bools  ints,
        NumberListToIntList boolVars, NumberListToIntList intVars,
        MaxSecurityLevel sMap (Cons (Bool, vars) stack) (D0 E) exprMax,
        IsSmallerThan minExpr exprMax minExprNeedsRaising,
        Bool.If minExprNeedsRaising exprMax minExpr minExpr') ⇒
  Expr Bool vars stack →
  (State (DSLState nextVar  unused  bools  ints  sMap  minExpr')
         (DSLState nextVar' unused' bools' ints' sMap' minExpr'') ()) →
  (State (DSLState nextVar  unused  bools  ints  sMap  minExpr)
         (DSLState nextVar' unused  bools  ints  sMap' minExpr)  ())
while cond body = ...
```

Finally, we alter the *buildWithInvariants* function so that it returns not only the AST of the While program, but also the security map, converted to a normal *Data.Map* value. Thus adding just two further parameters to the *DSLState*, and a few additional type class contexts where necessary, has allowed us to extend the type inference machinery to enforce a powerful and useful security model.

6 Related Work

This work arose out of the irritation of using DSLs without sufficient static checking. Although we have developed our own type-level libraries as necessary, most of the functionality and design is very similar to previous type-level programming work [4].

Structured Query Language (SQL) is probably both the most widely used and most reviled DSL. It is used to access database systems from within a program. The most common embedding of SQL is as a plain string resulting in convoluted programs that build up an SQL string as a result of user input. The SQL string is not statically checked, no invariants of the language are statically enforced, and frequently errors are only discovered at runtime when a database rejects an SQL string. SQL has seen some more advanced embeddings, such as the HaskellDB project [5, 6]. More recently, the Microsoft LINQ .Net extension [7] has created a type-safe general purpose query language which allows the programmer to query not only databases but in-memory data structures too.

The OpenGL Shading Language (GLSL) [8] is also widely used as it allows programs to be run directly on a GPU. GLSL is based on ANSI C and is typically dynamically compiled by the graphics card driver. Similarly to a naïve embedding of SQL, errors are left to be discovered at runtime.

It has been shown in [9] that by using Phantom Types (which we have made extensive use of in this work through GADTs), a monomorphic higher-order language can be embedded into Haskell in such a way that type soundness and completeness are achieved: i.e. an ill-typed program in the monomorphic higher-order language cannot be represented by the embedding and a well-typed program in the monomorphic higher-order language can always be represented by the embedding. Whilst our work offers no formal proofs, we do cater for polymorphic functions (equality), linear type inference (in that variables can change type), and work with an imperative language that is further away from Haskell than the λ-calculus as presented in their work.

Lava [10] is a DSL embedded in Haskell which caters for circuit design with support for simulation and verification of electronic circuits, in addition to traditional hardware design tools. The library takes advantage of Haskell's type classes to permit easy extension of the library and to enforce type safety. However, the basic type used is a representation of a bit or a number. Desired circuit invariants are supported through its verification mechanisms which makes use of external theorem provers. It would be very interesting to explore how our framework could allow circuit designers to enforce some invariants of their circuits without resorting to external theorem provers and what the advantages and disadvantages of each approach are.

7 Conclusions and Future Work

The convenience and power of embedding DSLs within a language make them an attractive and widely used technique. Statically enforcing their invariants

increases this appeal as it makes using a DSL more robust, reduces unexpected surprises for Haskell programmers, and makes designing and implementing a DSL less error prone. We have presented a flexible technique for lifting the invariants of a DSL into the Haskell type system and demonstrated how our framework can be used to statically enforce invariants relating to type checking and inference, and the Bell / LaPadula security model.

The analyses presented here have only required that the type of each statement depends on the type of the previous statement. Analyses which reach fixed points (for example, *live-variable* analysis), require that the type of a statement depends on the types of all preceding (or succeeding) statements as defined by the control flow graph rather than just the list of statements. This control-flow graph case is more complex, though we believe it is possible with our technique. We hope to explore these analyses in the future.

This work only uses functional dependencies as a means to perform type level computation. Associated data types and synonyms are a new addition to GHC and provide for type level computation but in a more flexible and less verbose way than functional dependencies. We hope to explore the use of associated data types and synonyms in the near future.

References

1. Peyton Jones, S.L., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP. (2006) 50–61
2. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)
3. Bell, D.E., Lapadula, L.J.: Secure computer systems: Mathematical foundations (1973)
4. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell, ACM Press (2004) 96–107
5. Leijen, D., Meijer, E.: Domain specific embedded compilers. SIGPLAN Not. **35**(1) (2000) 109–122
6. Bringert, B., Höckersten, A., Andersson, C., Andersson, M., Bergman, M., Blomqvist, V., Martin, T.: Student paper: Haskelldb improved. In: Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM (2004) 108–115
7. Pialorsi, P., Russo, M.: Programming Microsoft LINQ. Microsoft Press (2008)
8. Kessenich, J., Baldwin, D., Rost, R.: The opengl shading language (September 2006) http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf.
9. Rhiger, M.: A foundation for embedded languages. ACM Trans. Program. Lang. Syst. **25**(3) (2003) 291–315
10. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in haskell. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (1998) 174–184