Types and Analysis for Scripting Languages Exercises

Peter Thiemann

25.-27.2.2008

Here are some suggestions for projects relating to the topic of the lectures.

1 Soft Type Inference

This project is suitable if you have previous experience in implementing ML type inference.

Implement the soft-type inference engine for Mini-Scheme as defined in the Wright/Cartwright paper[WC97]. The following guidelines may be helpful.

- Define the required types
  - a `type_variable` is a pair of a list of type constructors (indicating the type constructors that may **not** be substituted for this type variable) and a reference to a (`sum_type option`)
  - a `sum_type` is either empty or a pair of a `type_component` and a `type_variable`
  - a `type_component` consists of a `type_constructor`, a `flag_variable`, and a list of `type_variable`
  - a `flag_variable` is a reference to a (`flag option`)
  - a `flag` is either `plus` or `minus`
  - a `type_scheme` contains a list of `type_variable`, a list of `flag_variable`, and a `type_variable` (with an empty list of type constructors)
  - a datatype `expr` for the expression syntax (`var`, `lam`, `app`, `chk_app`, `let`)
- Implement unification of two type variables (you may want to change the representation of `type_variable` to improve efficiency later on, but that's not strictly necessary).
- Implement type inference for the type system analogously to Milner's algorithm $\mathcal{W}$.

- Implement the soft typing translation as a second pass after type inference proper. First find out, what to do with the remaining free variables in the typing. Which of them can safely be instantiated to empty?

2 Flow Analysis and Abstract Interpretation

The idea of abstract interpretation is to run an interpreter on a non-standard, finite domain of values, the *abstract domain*. The abstract domain is constructed so that each abstract value corresponds to a set of values of the original domain. The operations on the abstract domain are approximations of the corresponding real operations, in the sense that the abstract value they produce always contains any possible real value.

Consider the following mini language with dynamic types. The base types are `boolean`, `number`, and `string`. There is a type constructor for `vector` t for vectors with elements of type t.

Expressions			
e	::=	x	variables
	|	c	constants
	|	$o(e, \ldots, e)$	primitive operations
Statements			
s	::=	$x = e$	assignment
	|	s;s	sequence
	|	`if` e `then` s `else` s	conditional
	|	`while` e `do` s	
Types			
t	::=	`boolean`	
	|	`number`	
	|	`string`	
	|	`vector` t	

The constants include numbers, `true`, `false`, and string constants. Among the primitive operations, there are the typical arithmetic, logic, and string operations, as well as

`make-vector` n v	creates vector of size n initialized with value v
`vector-ref` v i	vector access at position i (0-based)
`vector-set` v i w	overwrite position i in vector v with new value w

Each of the operations first attempts a dynamic conversion of its arguments to their expected types. If that fails, the operation raises a run-time error.

- Define the data types necessary to represent values in this language.
- Define suitable conversion operations between the data types. All of them should accept an arbitrary value and either return a value of the expected type or signal a conversion error.

- Write a standard interpreter for the language. Model the store as a mapping from variable names to values.
- Define a suitable notion of abstract values and implement a representation for abstract values. For simplification you may choose to omit arrays or just track the potential presence or absence of a value of a particular type. For technical reasons, the set of abstract values must be a finite complete lattice, for instance, a finite boolean algebra like a powerset of a finite set would work. The operations "get smallest element" and "least upper bound" are needed on a lattice.
- Define abstract versions of the primitive operations using the guideline given at the beginning of this project. Try to find the best possible abstractions that return the smallest possible abstract value.

  What are the correct abstractions of the conversion functions and where do they apply?
- Write an abstract interpreter. Model the store as a mapping from variable names to abstract values. Initialize each variable to $\bot$, the minimal element of the lattice of abstract values.

  Abstract evaluation of expressions proceeds in the obvious way. An assignment $x = e$ updates the store with the least upper bound of the abstract value of e and the previous abstract value stored for x. Abstract evaluation of a conditional visits both branches, unless the sublattice for boolean values can distinguish between `true` and `false`. Abstract evaluation of a while loop visits the body once unless e is proved to be `false`.

  The abstract evaluation of the whole program is repeated (*iterated*) until a fixpoint is reached. That is, until the mapping of variables to abstract values stabilizes.
- Run-time errors of a program can now be predicted by examining the uses of the conversion functions for failures. What exactly has to be tested?

References

[WC97] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.