Resolving Quartz Overloading

Oliver Pell and Wayne Luk

Department of Computing, Imperial College,
180 Queen's Gate London SW7 2AZ, UK
{op, wl}@doc.ic.ac.uk

Abstract. Quartz is a new declarative hardware description language with polymorphism, overloading, higher-order combinators and a relational approach to data flow, supporting formal reasoning for design verification in the same style as the Ruby language. The combination of parametric polymorphism and overloading within the language involves the implementation of a system of constrained types. This paper describes how Quartz overloading is resolved using satisfiability matrix predicates. Our algorithm is a new approach to overloading designed specifically for the requirements of describing hardware in Quartz.

1 Introduction

The term overloading, or ad-hoc polymorphism, describes the use of a single identifier to produce different implementations depending on context, the standard example being the use of "+" to represent addition of both integers and floating point numbers in most programming languages. Parametric polymorphism covers the case when a function is defined over a range of types but acts in the same way for each type, a typical example is the `length` function for lists. The functional language Haskell uses type classes [1] to combine overloading with parametric polymorphism using the Hindley/Milner type system [2].

Quartz is a new declarative hardware description language, intended to combine features found in the Pebble [3] and Ruby [4] languages. The language includes polymorphism with type inference and support for overloading, however previous approaches to combining type inference and overloading in software languages are not ideal for Quartz. This paper describes how Quartz overloading is resolved using a system of *satisfiability matrix predicates* which extend the Hindley/Milner type system to support overloading without using type classes.

Matrix predicates provide a generalisation of the basic type system that maintains full inference of types without any explicit definitions, in contrast to type classes which require explicit class and instance declarations.

2 Motivation

A Quartz description is composed of a series of blocks which are defined by their name, interface type, local definitions and body statements. A block's interface

D. Borrione and W. Paul (Eds.): CHARME 2005, LNCS 3725, pp. 380–383, 2005.

is divided, in a relational style, into a domain and a range. *Primitive blocks* represent hardware or simulation primitives and control the function of the circuit, while *composite blocks* contain statements which control the structure and inter-connections of the primitives.

Quartz has a simple but strong type system with three basic signal types for wires, integers and booleans. Quartz also supports both tuples and vectors of signals. The signal assignment operation "`=`" is overloaded to allow the assignment of static values to wires. Quartz blocks can be overloaded by defining multiple blocks with the same name, a mechanism that has a number of uses:

- Primitive blocks can be overloaded when multiple hardware primitives are available which essentially carry out the same operation e.g. a two-input adder and a constant-coefficient adder.
- Higher-order combinators can be overloaded when multiple blocks have the same basic function but slightly different parameterisations. It is sometimes useful to supply "hint" parameters to combinators to aid in the generation of parameterised output.
- Composite blocks can be overloaded with primitive ones as "wrappers" around the primitives e.g. if only a two-input adder primitive is available it may still be desirable to define an overloaded (composite block) constant-coefficient adder which instantiates the adder primitive appropriately.

In order to achieve this our general requirements, which differ substantially from typical software languages, are:

1. We have no interest in run-time polymorphism. We wish to eliminate polymorphism and overloading during elaboration.
2. We wish to minimise extensions to the syntax. Where possible overloading should be inferred, reducing designers' concerns so that they can work at a higher level of abstraction.
3. It is necessary to be able to express complex constraints between types in order to allow the overloading of blocks without a common type pattern.
4. It is necessary to support overloading of blocks with different numbers of parameters.
5. We can assume a *closed world* environment and have no need to support separate compilation since all libraries are expected to be available as source.

Evaluated against these requirements, type classes do not seem an appropriate mechanism for providing overloading in Quartz: although they support run-time polymorphism, this is not useful; the language must be extended with extensive class and instance declarations; single-parameter type classes (as in the Haskell specification) can not express complex constraints between types and while multi-parameter type classes can type inference is then undecidable; they do not easily support overloading blocks with different numbers of parameters; and inferred types are sometimes ambiguous due to the *open world* assumption.

To meet these requirements, we use a system based around a language of satisfiability matrix predicates – matrices that represent possible values of a type and relationships between type variables. This system minimises ambiguity and can express n-ary constraints between type variables clearly and easily.

(a) Instances of R

$R_1 : \; bool\ int \sim int$

$R_2 : \; \sigma\ wire \sim \sigma$

$R_3 : \; int\ int \sim wire$

(b) Matrix π^R

$$\begin{pmatrix} bool & int & int \\ \sigma & wire & \sigma \\ int & int & wire \end{pmatrix}$$

(c) Statements

$a\ b\ ;\ R\ ;\ c.$

$c\ d\ ;\ R\ ;\ e.$

(d) Instantiations

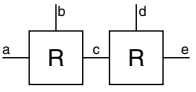

Fig. 1. Multiple instance types can be represented as a satisfiability matrix

(a)

$$\begin{pmatrix} - & - & - \\ int & wire & int \\ int & int & wire \end{pmatrix}$$

(b)

$$\begin{pmatrix} int\ wire & int & wire & int \\ int\ wire & int & int & wire \\ int & int & wire\ wire & wire \end{pmatrix}$$

(c)

$$\begin{pmatrix} int\ wire & int & wire & int \\ - & - & - & - \\ int & int & wire\ wire & wire \end{pmatrix}$$

Fig. 2. Evolution of a predicate matrix during type checking

3 Satisfiability Matrix Predicates

Our system is implemented as a conservative extension to Robinson's unification algorithm [5] to support satisfiability matrices. We will introduce our system with an example. Suppose there are multiple, overloaded instances of a block R with types as in Fig. 1(a) (where σ is a polymorphic type variable). The types of these three instances can be represented as a predicate matrix π^R as shown in Fig. 1(b) where each row of the matrix contains the type for an instance and there is a column for each argument position.

Suppose then it is desired to type check the two Quartz statements in Fig. 1(c), which instantiate two R blocks as shown in Fig. 1(d), where a is known to have type *int*, b has unknown type β, c has unknown type γ and d and e have type *wire*.

Because there are two instantiations of the R block which could have different types, two matrices π^R and $\pi^{R'}$ will be used during type checking. The inference process attempts to unify each argument type with the appropriate column in the matrix. Type checking the first statement involves three unification operations: $unify(int, \pi_0^R)$, $unify(\beta, \pi_1^R)$ and $unify(\gamma, \pi_2^R)$ (where subscripts indicate the column number).

Unifying a type with a matrix column involves unifying that type with every element in the column. If this operation generates a substitution within the matrix, this substitution is applied along that row of the matrix. If a column element does not unify then that row is removed from the matrix. The result of the three unification operations above is shown in Fig. 2(a), note that the first row did not match and has been removed while *int* has been substituted for the unknown type σ. The operations have bound the type variables β and γ into the matrix: $\{\beta \mapsto \pi_1^R, \gamma \mapsto \pi_2^R\}$.

When type checking the second statement the type of c (γ) must be unified with $\pi_0^{R'}$ however it is already bound into the first matrix so matrices π^R and $\pi^{R'}$ must be merged. This produces a single matrix, shown in Fig. 2(b), with one row for each valid possible combination of types from the two source matrices where

the two columns bound to type γ could be unified. Type checking continues by unifying the type of d (*int*) across the appropriate matrix column, which was $\pi_1^{R'}$ but is now π_3^R in the new matrix. The type *int* does not match with all elements in the column and so one row is eliminated as shown in Fig. 2(c).

Finally the type of signal e (*wire*) is unified with π_4^R which matches a single matrix row. The overloading is resolved with the type mapping $\{\beta \mapsto int, \gamma \mapsto wire\}$. The first R block is selected as R_3 with type $int\ int \sim wire$ and the second R block is selected as R_2 with type $wire\ wire \sim wire$.

The case when a type constructor with unknown type variables within it, such as the tuple (ϕ, ψ), is unified with a matrix column needs to be handled separately. The full type is unified across the original column while new columns are generated in the matrix for the unknown type variables to be bound to. The operation $unify\big((\phi, \psi), \pi_0^R\big)$ applied to the original π^R would return the substitution $\{\phi \mapsto \pi_3^R, \psi \mapsto \pi_4^R\}$ and the matrix would be left with a single row of $\big((\phi', \psi')\ \ wire\ \ (\phi', \psi')\ \ \phi'\ \ \psi'\big)$ where the tuple has been substituted for σ.

Satisfiability matrices can also support blocks with different numbers of parameters by extending the Quartz type system with an empty/void type Ω, which can be used to "pad" matrices and block types so that they are all the same length. Ω only unifies with itself so blocks with the wrong number of parameters are eliminated from the matrix when unification fails.

During type checking predicate matrices grow (due to mergers) and shrink (due to non-matching rows being eliminated) before reaching a point where overloading can be resolved. It is often possible to substantially optimise matrices to reduce their size, for example by merging identical columns.

4 Conclusion

Satisfiability matrices permit the expression of complex n-ary constraints between types and the overloading of Quartz blocks without extending the language syntax. We believe our system is superior to type classes for overloading in Quartz and other similar hardware description languages. Future work includes investigating the theoretical properties of satisfiability matrices and developing optimisation strategies to minimise the amount of matrix data stored.

References

1. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. POPL '89, ACM Press (1989) 60–76
2. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17** (1978) 348–375
3. Luk, W., McKeever, S.: Pebble: a language for parameterised and reconfigurable hardware design. In Proc. FPL'98. LNCS 1482, Springer-Verlag (1998) 9–18
4. Jones, G., Sheeran, M.: Circuit design in Ruby. In Staunstrup, J., ed.: Formal Methods for VLSI Design, North-Holland/Elsevier (1990) 13–70
5. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12** (1965) 23–41