Facilitating Reuse in Hardware Models with Enhanced Type Inference

Manish Vachharajani Neil Vachharajani Sharad Malik David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544
{manishv, nvachhar, malik, august}@princeton.edu

ABSTRACT

High-level hardware modeling is an essential, yet time-consuming, part of system design. However, effective component-based reuse in hardware modeling languages can reduce model construction time and enable the exploration of more design alternatives, leading to better designs. While component overloading and parametric polymorphism are critical for effective component-base reuse, no existing modeling language supports both. The lack of these features creates overhead for designers that discourages reuse, negating any benefits of reuse.

This paper presents a type system which supports both component overloading and parametric polymorphism. It proves that performing type inference for any such system is NP-complete and presents a heuristic that works efficiently in practice. The result is a type system and type inference algorithm that can encourage reuse, reduce design specification time, and lead to better designs.

Categories and Subject Descriptors:
D.3.3 Programming Languages: Language Constructs and Features–*Polymorphism*
I.6.2 Simulation and Modeling: Simulation Languages

General Terms: Algorithms, Design, Languages

Keywords: Liberty Simulation Environment (LSE), component reuse, polymorphism, component overloading, type inference

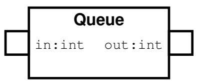
1. INTRODUCTION

Exploring the design space when designing a hardware system is vital to realizing a well-performing design. Hardware complexity has made building high-level system models to explore the design space an essential, yet burdensome and time-consuming, part of system design. Since overall design quality improves as more designs are explored, reducing the time to develop system models can dramatically improve design quality. To reduce development times, others have proposed high-level component-based modeling systems supporting reusable components [3, 9, 11]. Reuse, however, is effective only if components are flexible enough to be used in a wide range of designs and if using flexible components does not have an overhead so high it precludes their use in practice [11].

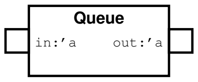
A common mechanism to increase the flexibility of components is to use programming language techniques such as polymorphism. Consider, for example, a component library that contains various ALU models to handle different data types, such as integers and several formats of floating point numbers. While it is possible for the user to specifically select which ALU is appropriate for a processor model being built, forcing the user to make this selection can be cumbersome. Instead these components could be combined into a single overloaded component. This *component overloading* (also known as ad-hoc polymorphism) allows the system to automatically select an appropriate component implementation by analyzing the relationship of the overloaded component's supported data types to the data types of components connected to it. This frees designers from dealing with less important details regarding typing and allows them to focus on high-level system design issues. Some existing hardware modeling systems, such as Balboa [4], support this type of polymorphism.

Component overloading reduces the overhead of using a component, but it still requires the implementation of several ALU behaviors, one for each supported data type. A separate class of components, such as queues, memories, and crossbar switches, have high-level functionality that is independent of the data types they manipulate. *Parametric polymorphism* frees designers from unnecessary reimplementation of these components by providing a mechanism to build a single implementation for use with many data types. A familiar example of this type of polymorphism is a SystemC [9] component built using a C++ template. In general, a parametrically polymorphic component uses *type variables* (in C++, template parameters) in place of concrete types in its definition. When such a component is instantiated, the user provides a concrete type for each type variable thus resolving the polymorphism. The compiler uses this information to automatically specialize the component behavior to handle the specified concrete types. For example, a queue model using parametric polymorphism could be instantiated in one part of a system to queue instructions and the *same* model could be instantiated elsewhere to queue Ethernet packets. The compiler would specialize the queue's behavior to handle the different data types without any user intervention.

Unfortunately, the excessive type instantiations needed when using many parametrically polymorphic components often discourages their use in practice, thus nullifying the model development time benefits they provide. Just as in the case of component overloading, it is possible to build a system that automatically identifies type variable values [15] by using the system connectivity to identify what the values for the type variables should be. Clearly, if



(a) Monomorphic Queue.



(b) Polymorphic Queue.

```
template
class queue<T> {
  // input port
  sc_in<T> in;
  // output port
  sc_out<T> out;
  ...
}
```

(c) SystemC code.

Figure 1: Port interface for a simple queue.

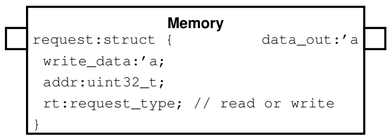


Figure 2: Port interface for a simple memory.

the output of an instruction fetch unit is connected to a queue, the queue will be storing instruction data and not Ethernet packets.

Unfortunately, building a modeling system which automatically infers type instantiations for parametrically polymorphic components while also selecting implementations of overloaded components is non-trivial. In this paper, we propose a type system for high-level hardware modeling systems that supports both parametric polymorphism and component overloading. We show that the type inference problem for this type system is NP-complete. We then present a heuristic type inference algorithm that automatically infers the type instantiations for parametrically polymorphic components and automatically selects the appropriate implementation for overloaded components. We show that this algorithm has reasonable run-times for models seen in practice.

The remainder of this paper is organized as follows. Section 2 describes the type system, including examples of how the features may be used. Section 3 describes how this type system can be applied to existing systems. Section 4 presents the type inference problem for this type system and a proof of its NP-completeness. Section 5 presents our heuristic type inference algorithm and Section 6 presents an evaluation of the type system and inference algorithm. Section 7 summarizes the contributions and concludes.

2. THE TYPE SYSTEM

In this section, a type system that supports both component overloading and parametric polymorphism will be constructed by examining the common usage paradigms for these features. The type system is presented in the abstract since it can be used with many different modeling languages. For clarity, the examples occasionally compare these abstract concepts to concepts in SystemC.

Consider a very simple queue whose port interface is shown in Figure 1(a). In the interface definition of this component, one would like to specify an `in` port and an `out` port[1]. Data received on the `in` port is enqueued, the item at the head of the queue is transmitted on the `out` port and dequeued. As shown, this queue component can only handle `int`s (denoted by the `:int` in the declaration). Instead of `int`, the queue could have been written to support other basic data types such as `bool`, arrays, structures, or lower-level types such as bits or `ieee_std_logic_vector`.

Since the high-level behavior of the queue is independent of the data type stored in it, the queue is an ideal candidate for parametric polymorphism. The interface for a polymorphic queue component replaces the explicit type specified on the `in` and `out` ports with a type variable. This is shown in Figure 1(b). In the example, this type variable is denoted `'a` (type variables are prefixed with a `'`) and is common to both ports. Since the `in` and `out` ports share the same type variable, they are constrained to have the same type. The interface shown in the figure is equivalent to the SystemC code shown in Figure 1(c) where the template argument `T` takes the place of type variable `'a`.

[1]Clock and control signals are omitted from the diagrams for simplicity. In fact, some high-level modeling systems abstract these interfaces away [14].

From this example, we see that our type system should have type variables so that ports, state, etc. in a model can be parametrically polymorphic. Additionally, the type system should allow more complex data types, such as structures, to be built using type variables. For example, consider a memory component, as shown in Figure 2. This component could have a request port to access memory and output data port to return the results of read requests. As shown, the `request` port needs to have some fields with fixed types for addressing and identifying the type of request, and then a data field, whose type is specified with a type variable, to carry the data for any write requests. Just as before, this same type variable can be used on the `data_out` port to ensure that the data written to the memory has the same type as the data that is read out. From this example it is clear that one should be able to use a type variable wherever one could use a regular type in a port interface definition.

The legal types in this type system are described by the following grammar:

Basic Types	τ	::=	$\texttt{int} \mid \ldots \mid \tau[n] \mid$
			$\texttt{struct}\{i_1 : \tau_1; \ldots i_n : \tau_n;\}$
Type Variables	α, β, γ	::=	$'identifier$
Type Schemes	τ^*	::=	$\alpha \mid \tau \mid \tau^*[n] \mid$
			$\texttt{struct}\{i_1 : \tau_1^*; \ldots i_n : \tau_n^*;\} \mid$
			$(\tau_1^* \mid \ldots \mid \tau_n^*)$

Here, the basic types are the standard programming language types (the $\tau[n]$ notation is an array of the given type) that would be communicated over ports at model runtime. The type schemes (other than the last one) are the collection of basic types, type variables, and basic types containing references to type variables. Given a basic type for all unbound type variables in any these type schemes, we get a valid basic type that is an instantiation of the type scheme. As will be shown, the last type scheme, the *disjunctive type scheme*, is used to support component overloading.

To specify component interfaces for overloaded components, a port on the interface is annotated with a disjunctive type scheme. These type schemes enumerate a list of type options. When a component is instantiated, one of these options is *statically* selected and becomes the type of the entity which carried the disjunctive type scheme. This *disjunctive* type scheme is denoted as `type1 | ...| typen`. The following rule is added to the type system's grammar to allow this type scheme. Note that the disjunctive type scheme is different than a union type, since union-typed entities may have a value of one type from a pre-specified list, but this type may change at run time. Accordingly, union types do not facilitate component overloading.

To understand how this type scheme facilitates component overloading, recall the overloaded ALU discussed in Section 1. Assuming the overloaded component supported both the `int` and `float` data type, then, in this type system, one would assign a common

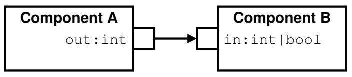


Figure 3: Connected Components

type variable to the ALU's input and output ports and then constrain the type variable to be equal to `int | float`.

3. EXISTING MODELING SYSTEMS

In this section we show how the previously described type system can represent the port-interface type systems in some existing high-level modeling languages. The section will also discuss what forms of type inference the systems support if any.

SystemC. SystemC is a structural modeling system built as a library for C++. Components are C++ objects and component interface definitions are in the corresponding C++ classes. Ports and connections are created using the SystemC library. SystemC supports parametric polymorphism through the use of C++ templates, but requires manual resolution of this polymorphism rather than using type inference. SystemC does not support component overloading, nor does it recognize subtyping when considering if two ports have the same type.

The template arguments for a SystemC component correspond to type variables. Basic C++ data types correspond to the basic types presented earlier. Template classes' templatized port interfaces can be characterized using the type schemes presented. Classes can be represented by adding function pointers to the basic types and representing classes as `struct`s that contain function pointers.

Balboa. Balboa is a framework to compose components built in otherwise incompatible systems, including SystemC components. Components are connected and their interface specified via an interface definition language (IDL). The IDL allows overloading where each component has a set of legal types for its port interface. Each legal interface has a corresponding implementation of the component for use during execution. Before beginning simulation, the system performs type inference to select the appropriate implementation based on the port connectivity. However, Balboa's type inference algorithm does not support parametric polymorphism since the IDL does not support it (parametrically polymorphic SystemC components must be preinstantiated with a fixed range of types before being used with Balboa).

Basic IDL types in Balboa correspond to the basic types in the type system presented in Section 2. Overloaded components' port interface signatures can be constructed using the same method described previously for the overloaded ALU.

Ptolemy II. Ptolemy II is a modeling environment designed to support multiple models of computation. It has a type system that supports type inference in the presence of parametric polymorphism and subtyping, but unfortunately, it does not support component overloading [15].

Types in Ptolemy II correspond to types in Section 2's type system similarly to SystemC. Section 2's type system does not handle subtyping, but can be extended to do so.

4. THE TYPE INFERENCE PROBLEM

To automatically resolve the polymorphism for the type system described in Section 2, a type inference algorithm must identify a basic type for all type variables and ensure that a single type is selected from every disjunctive type scheme. When making such selections, the algorithm must respect each port's type scheme and ensure that connected ports share a common type. For example, consider the two components in Figure 3. Here, the `in` port on component B has the type scheme `int | bool`. Type inference must assign it the basic type `int` or the basic type `bool`. The port `out` on component A has type scheme `int` and thus can only be assigned type `int`. Furthermore since `in` and `out` are connected, they must have the same type. In this case, the type inference algorithm, because of the equality constraint, must assign basic type `int` to port `in`. Since `int` is a basic type that corresponds to the type scheme `int | bool`, all conditions are satisfied and a valid, unique type assignment has been found.

If no valid type assignment is possible (e.g. if the `in` port on component B has type scheme `float | bool`) then the system is said to be *over-constrained*. It is also possible to have a system with multiple valid type assignments (e.g. if the `out` port on component A also had the type scheme `int | bool`). Such systems are said to be *under-constrained*. Over-constrained or under-constrained systems are considered malformed systems and the type inference algorithm that will be presented will report an error if such systems are encountered.

The type inference problem can be formalized as solving a set of constraints involving equality of type schemes. First, a type variable is created for every port in a system. Second, a core set of constraints is formed stating that these new type variables must be equal to the type scheme for the respective port. Third, the core set of constraints is augmented with constraints formed by equating the type variables for any two ports that are connected. The legal constraints in the type system are given by the following grammar:

$$\text{Constraints} \quad \phi \quad ::= \quad \top \mid \tau_1^* = \tau_2^* \mid \phi_1 \wedge \phi_2$$

$\top$ is the trivially true constraint, $\tau_1^* = \tau_2^*$ is a constraint asserting the equality between two type schemes, and $\phi_1 \wedge \phi_2$ is the conjunction of the sub-constraints ϕ_1 and ϕ_2.

The type inference engine must resolve all the type schemes to basic types and assign basic types to all type variables according to the constraints. The constraint is constructed and evaluated *at once* for the *entire model*. Thus, the inference engine may reject constraints with multiple solutions (i.e. an underconstrained system) since the model is ill-defined in this case. Constraints with no solution obviously correspond to a malformed model.

THEOREM 1. *The type inference problem is NP-complete.*

PROOF. A sketch of the proof showing the problem is in NP is as follows. For each disjunctive type scheme, non-deterministically choose which of the type schemes in the disjunction to use. After this resolution of disjunctive type schemes, the problem reduces to that of unification which is in P [10]. Thus the type inference problem is in NP.

To show the problem is NP-hard, we show how to reduce any instance of the monotone 1-in-3 SAT problem, a known NP-complete problem [5], to the type inference problem. 3-SAT is a SAT problem where one must decide if there exists an assignment of truth values to a set of boolean variables, B, such that each clause in a set of disjunctive clauses, C, with exactly 3 literals per clause, is satisfied. Monotone 1-in-3 SAT is a 3-SAT problem where each member of B only appears in non-negated form and only one literal in each clause may have a truth value of '1'.

The reduction proceeds as follows. Let the type `int` correspond to the truth value '1', and let `bool` correspond to '0'. For each variable $b_i \in B$ create a type variable $\alpha_{b,i}$. For each clause, $c \in C$, create a type variable α_c. Each clause has the form (b_i, b_j, b_k) where $b_i, b_j, b_k \in B$. The only satisfying assignments for each clause are $S_0 = (1,0,0)$, $S_1 = (0,1,0)$, and $S_2 = (0,0,1)$. Let

$'S_0$ = `struct { x:int; y:bool; z:bool;}`
$'S_1$ = `struct { x:bool; y:int; z:bool;}`
$'S_2$ = `struct { x:bool; y:bool; z:int;}`

For each clause add the constraint $\alpha_c =' S_0 \mid 'S_1 \mid 'S_2$. This constraint ensures that each clause has a legal satisfying value, S_0, S_1, or S_2. To ensure that boolean variables in the clause get the appropriate value based on which satisfying assignment is chosen, add the following constraint for each clause:

$$\alpha_c = \texttt{struct \{ x:}\alpha_{b,i}\texttt{; y:}\alpha_{b,j}\texttt{; z:}\alpha_{b,k}\texttt{;\}}$$

If the inference problem for this set of constraints has a solution, the corresponding monotone 1-in-3 SAT problem is satisfiable. If not, the problem is unsatisfiable. Thus, solving the inference problem solves the corresponding monotone 1-in-3 SAT problem. □

The type system and inference problem presented here is very similar to the type system and inference problems in languages such as Haskell. However, the Haskell problem is undecidable in general [13]. There exist restricted versions of the type system that are decidable [8, 12]. Unfortunately, of these, the restrictions that yield acceptable computational complexity [8] are not desirable in a structural modeling environment since they forbid common port interface typings. For other restricted versions of the type system, we know of no heuristic algorithms that are appropriate for instances of the problem that arise in the structural modeling domain.

The type inference problem also seems easily mapped to SAT. However, a straight-forward mapping with a bounded width binary encoding for each type that is uniform across all problem instances is not possible. This is because one can arbitrarily nest `struct`s and arrays leading to an unbounded number of types across all problem instances. It should be possible to identify a finite set of types that can occur in a particular problem instance, however, this is also non-trivial and left to future work.

5. THE INFERENCE ALGORITHM

The substitution algorithm used for type inference in languages such as ML [7] can be extended for the presented type system by modifying the algorithm to handle the disjunctive type scheme. This section presents such an extension and proves the correctness of the approach. The extended algorithm generates a typing context that maps type variables to basic types for all satisfiable constraints. Since the simplest extension of the algorithm has prohibitively large run times, this section will also present heuristics that make run-times reasonable for models seen in practice.

5.1 Basic Algorithm

The basic ML-style substitution algorithm works by simplifying each term in the constraint, ϕ, and then eliminating it. As the constraints are simplified, the algorithm will create a new simpler constraint and then recursively process this new constraint. During this simplification, the algorithm builds up a typing context, T, that maps type variables to type schemes. The recursion occurs based on the structure of the constraint. Therefore, there must be a rule that handles each production in the grammar defining the constraint. The typical rules are shown in Table 1. If the constraint has the form shown in the left column of the table, then the action on the right hand column is taken (only the first matching rule, starting from the top of the table, is applied). Each of these rules operates by simplifying the constraint based on the definition of equality for the type schemes. Recall that τ denotes basic types, τ^* denotes type schemes, and $\alpha, \beta, \gamma, ...$ denote type variables. The notation $[y/x]Z$ means: substitute every occurrence of y in Z with x. Note that the state of the algorithm at any step can be summarized by the pair (T_i, ϕ_i) where T_i is typing context and ϕ_i is the simplified constraint. The initial state is $(\emptyset, \phi)$.

The disjunctive constraint requires a new, atypical, rule and a modification to the form of constraints, since disjunctive type schemes are forbidden from the rule in the third row of the table. The rule on the third row of the table forbids disjunctive type schemes to ensure that the typing context, T, always maps type variables to concrete types, if all type variables had values. Allowing the disjunctive type scheme in the typing context would violate this property. We can avoid this difficulty by replacing every disjunctive type scheme with a type variable and adding a constraint asserting that the type variable is equal to the disjunctive type scheme. This simplified constraint involving disjunctive type schemes is handled by the following new rule. If the constraint is of the form $(\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi$, then n inference problems are created each starting in the state $(T_{\text{old}}, \phi_i \wedge \phi)$, where $\phi_i \equiv (\tau^* = \tau_i^*)$. Each problem is solved by recursively applying the algorithm. When the algorithm terminates for each subproblem, it reports that $\phi_i \wedge \phi$ was unsatisfiable or returns a typing context T_i. If, $\forall i, j,\ 1 \le i, j \le n$, $\phi_i \wedge \phi$ and $\phi_j \wedge \phi$ were satisfiable and $T_i \ne T_j$, then the system is under-constrained. The algorithm terminates if this is detected since only unique solutions are acceptable.

For the extended algorithm to be correct, two things must be true. First, the algorithm must terminate. Second, the algorithm must incrementally build a solution to the inference problem in each step. The next two theorems formally state these two properties.

THEOREM 2. *The algorithm's state sequence is finite.*

PROOF. To each state (T, ϕ), we will assign a pair (n, s), where n is the number of type variables in ϕ and s is the sum of the number of type schemes in ϕ. If (T_i, ϕ_i) is satisfiable, each rule shown in Table 1 takes (T_i, ϕ_i) and converts it into a state (T_{i+1}, ϕ_{i+1}) where $n_{i+1} < n_i$ or both $n_{i+1} = n_i$ and $s_{i+1} < s_i$. Consider each row of the table. The first row, removes two type schemes from ϕ_i. The third row reduces the number of type variables by one. The fourth row, removes two type schemes from ϕ_i, as does the fifth row. All rows decrease either s (without increasing n) or decrease n. Thus, the sequence of states form a strictly decreasing sequence of ordered pairs of integers (the pairs (n, s)). Since the sequence is lower bounded ($n \ge 0$ and $s \ge 0$), the sequence will converge in a finite number of steps.

If some state in the sequence is unsatisfiable, then the algorithm terminates, and thus the theorem is trivially true.

Upon encountering a disjunctive constraint, the algorithm recursively applies itself to many subproblems. Each subproblem contains one less type scheme and thus, by the metric introduced earlier, is simpler than the current problem. Therefore, by induction, no subproblem will lead to an infinite sequence of states. Thus, the algorithm's state sequence is finite. □

The next theorem will show that the final typing context T provides a solution to the initial constraint, however some useful notation will be introduced first. The application of a typing context, S, to a type scheme, ϕ involves recursively substituting all the type variable to type scheme mappings in the type context S into the type schemes of ϕ. This operation will be denoted by $\overline{S}(\phi)$. Next, we will write $S \models \phi$ if the typing context S satisfies the constraint

Constraint form	Operation
$(\tau^* = \tau^*) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}, \phi_{\text{new}} \leftarrow \phi$
$(\alpha = \texttt{struct \{ x1:}\alpha\texttt{; }\ldots\texttt{xn:}\tau^*_n\texttt{;\}}) \wedge \phi$	System is unsatisfiable
$(\alpha = \tau^*) \wedge \phi$, τ^* contains no disjunctive type schemes	$T_{\text{new}} \leftarrow [\alpha/\tau^*]T_{\text{old}} \cup (\alpha \mapsto \tau^*)$, $\phi_{\text{new}} \leftarrow [\alpha/\tau^*]\phi$
$(\tau^*_1\texttt{[n]} = \tau^*_2\texttt{[n]}) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}, \phi_{\text{new}} \leftarrow (\tau^*_1 = \tau^*_2) \wedge \phi$
$(\texttt{struct \{ x1:}\tau^*_{1,1}\texttt{; }\ldots\texttt{; xn:}\tau^*_{1,n}\texttt{;\}} = \texttt{struct \{ x1:}\tau^*_{2,1}\texttt{; }\ldots\texttt{; xn:}\tau^*_{2,n}\texttt{;\}}) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}$, $\phi_{\text{new}} \leftarrow (\tau^*_{1,1} = \tau^*_{2,1}) \wedge \ldots \wedge (\tau^*_{1,n} = \tau^*_{2,n}) \wedge \phi$
$\top$	Constraint satisfiable, Solution in T

Table 1: Simple Substitution Algorithm

ϕ. Formally, this relation is inductively defined as follows:

$$\begin{aligned} S &\models \top && \text{iff} \quad \text{always} \\ S &\models \tau^*_1 = \tau^*_2 && \text{iff} \quad \overline{S}(\tau^*_1) \cap \overline{S}(\tau^*_2) \neq \emptyset \\ S &\models \phi_1 \wedge \phi_2 && \text{iff} \quad S \models \phi_1 \text{ and } S \models \phi_2 \end{aligned}$$

In the second rule, we treat type schemes like sets. The type scheme $\tau^*_1 \mid \ldots \mid \tau^*_n$ is considered to be the set $\{\tau^*_1, \ldots, \tau^*_n\}$. Other type schemes are considered to be singleton sets. Finally, given a state (T, ϕ), we call S a solution for (T, ϕ) if $S = T \cup U$ and $S \models \phi$ for some U whose domain is disjoint from the domain of T.

THEOREM 3. *If the algorithm transitions from (T, ϕ) to (T', ϕ'), then S is a solution for (T, ϕ) iff S is a solution for (T', ϕ').*

PROOF. We will prove this statement by induction on the structure of the transitions. The transitions presented in Table 1 are almost identical to those for programming languages such as ML and the truth of this theorem is well known for those transitions [6].

For a transition based on the disjunctive rule, we have $\phi = (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi''$ and $\phi' = \top$. Assume that S is a solution to the state $(T, (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi'')$. Since $S \models (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi''$, then $S \models \tau^* = \tau^*_i \wedge \phi''$ for some i by the definition of the $\models$ relation. Therefore S is also a solution to the state $(T, \tau^* = \tau^*_i \wedge \phi'')$. The algorithm will transition in many steps from the state $(T, \tau^* = \tau^*_i \wedge \phi'')$ to $(T', \top)$. By the inductive hypothesis, S is also a solution to the state $(T', \top)$.

Now, assume that S is a solution to the state $(T', \top)$. We must show that S is also a solution to the state $(T, (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi'')$. Without loss of generality, we will assume that the algorithm will transition in many steps from the state $(T, \tau^* = \tau^*_1 \wedge \phi'')$ to $(T', \top)$. By the inductive hypothesis, we have that S is a solution to the state $(T, \tau^* = \tau^*_1 \wedge \phi'')$. This implies that $S = U \cup T$ for some disjoint U and that $S \models (\tau^* = \tau^*_1 \wedge \phi'')$. By the definition of the $\models$ relation, this implies that $S \models (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi''$. By the definition of solution, we have that S is a solution to the state $(T, (\tau^* = \tau^*_1 \mid \ldots \mid \tau^*_n) \wedge \phi'')$. $\square$

Theorem 2 proves that the algorithm always terminates. Further Theorem 3 proves that each step of the algorithm constructs an incremental solution to the inference problem. Therefore, if the algorithm terminates with the constraint equal to $\top$, then the typing context forms a solution to the initial state and thus to the initial type inference problem. When the algorithm terminates with a constraint other than $\top$, the terminating constraint is over-constrained and thus by Theorem 3 so too is the initial state. Finally, we also reject states where we detect *multiple* solutions. Theorem 3 tells us that these multiple solutions are all valid for the initial constraint also. Therefore, the initial constraint system is under-constrained.

5.2 Heuristics

The basic algorithm just described has prohibitively large run times, which is not surprising given that the inference problem is NP-complete. The large run-times are due to the subproblem exploration required by the disjunctive constraints. This section describes an additional rule and two heuristics, inspired by heuristics for the DLL SAT algorithm [2], which dramatically reduce the number of subproblems that need to be solved, which in turn drastically reduces run-time. Note that these heuristics only affect run-time, not the quality or correctness of the typing context produced.

5.2.1 Disjunctive Constraint Simplification

The additional rule simplifies disjunctive constraints without creating and solving subproblems. Consider a constraint of the form $((\tau^*_{1,1} \mid \ldots \mid \tau^*_{1,n}) = (\tau^*_{2,1} \mid \ldots \mid \tau^*_{2,n})) \wedge \phi$. This constraint can obviously be simplified by eliminating type schemes, $\tau^*_{1,j}$, that do not have a compatible type scheme in $\tau^*_{2,k}$, and vice versa. Recognizing all incompatible type schemes is equivalent to the original inference problem, but certain incompatibilities are easy to verify. For example, any basic type τ is only compatible with type schemes τ and α. Two `struct` type schemes with differing element names or with differing numbers of elements are also incompatible. Similar rules can be constructed for arrays.

5.2.2 Processing Constraints Out-of-order

The first heuristic to reduce the number of subproblems to be solved delays the creation of subproblems due to disjunctive constraints as long as possible. This allows terms that do not contain disjunctive type-schemes (disjunctive terms) to be simplified *before* creating subproblems to address a disjunctive term. These simplifications may make the disjunctive-constraint-simplification rule applicable, eliminating the subproblem creation entirely, or significantly reduce the size of the subproblems by solving common terms in the constraint only once.

The proposed heuristic to accomplish this causes the algorithm to evaluate the simplification rules in multiple phases. In the first phase, the constraint terms are treated as a list, and each constraint term is simplified in-order. If any constraint has a leading disjunctive term, the constraint is reordered to delay evaluation of the disjunctive term. This iteration over constraint terms proceeds until no simplifications can be made without handling a disjunctive term via subproblem creation. In the second phase, the leading disjunctive term is handled via the subproblem creation used in the simple algorithm.

5.2.3 Partitioning the Constraint

The second heuristic leverages the independence of subproblems in type inference algorithm. Consider the following constraint:

$$('\texttt{a} = \texttt{int}|\texttt{bool}) \quad \wedge \quad ('\texttt{a} = \texttt{int}|\texttt{char}) \wedge \qquad (1)$$
$$('\texttt{b} = \texttt{int}|\texttt{bool}) \quad \wedge \quad ('\texttt{b} = \texttt{int}|\texttt{char}) \wedge \qquad (2)$$
$$('\texttt{c} = \texttt{int}|\texttt{bool}) \quad \wedge \quad ('\texttt{c} = \texttt{int}|\texttt{char}) \qquad (3)$$

In this case there are six disjunctive type schemes with two possible types, thus the basic algorithm would need to solve $2^6 = 64$

subproblems. The heuristics presented thus far would reduce this to $2^3 = 8$ since half of the disjunctive type schemes can be handled by disjunctive constraint simplification after subproblems are created for the other half of the disjunctive type schemes. The number of subproblems can be reduced further still by recognizing that constraint terms given by (1), (2), and (3) each refer to disjoint sets of type variables and can thus be solved independently. Applying this technique would yield $2^2 + 2^2 + 2^2 = 12$ subproblems without disjunctive constraint simplification or $2 + 2 + 2 = 6$ subproblems with disjunctive constraint simplification.

In general, any constraint can be partitioned into multiple constraints by placing terms of the original constraint into different sets such that if two constraints are in different sets, they refer to non-overlapping sets of type variables. Each of these sub-constraints can be solved separately since its solution is only affected by mappings for type variables that it references. After performing such a partitioning, if any of the sub-constraints are unsatisfiable, then the whole system is unsatisfiable since no typing context which contains mappings for the type variables in the sub-constraint will solve the original constraint. If all the sub-constraints are satisfiable, a union of each sub-constraint's typing context is the typing context that solves the original constraint. In general the basic algorithm will need to explore $m \cdot n$ subproblems if it contains a disjunctive type scheme with m options and a disjunctive type scheme with n options. If these type schemes are independent, after partitioning, only $m + n$ subproblems need be explored. Thus, the savings due to partitioning increases exponentially with the number of independent disjunctive constraints. Partitioning the constraint after reordering but before attempting to solve any subproblems increases the likelihood of finding independent disjunctive type schemes and is the best place to perform partitioning.

6. EXPERIMENTAL RESULTS

The type system and inference algorithm described in this paper are implemented in the Liberty Simulation Environment (LSE) [14], a publicly available high-level hardware modeling tool. This section measures the effectiveness of this work by evaluating several LSE models created for other research and instruction.

To evaluate the effectiveness of type inference in easing the use of polymorphism, the number of explicit type instantiations with type inference is compared to that without type inference. The results are shown in Table 2. As can be seen from the table, far fewer instantiations are needed with type inference thus significantly reducing the burden on the user. In two of the models studied, only 8 out of approximately 100 type instantiations were user specified. This shows how type inference can nearly eliminate any overhead associated with polymorphism while preserving all of its benefits.

To determine if the inference algorithm presented is practical, the run times of the algorithm with and without the heuristics presented in the previous section were measured. Table 2 also presents these results. The model specification language compiler and type inference algorithm are implemented in Java. The run times were measured using Sun's Java VM 1.4.1_02 on a 3.0GHz Pentium 4 machine with 2 GB of RAM running RedHat Linux 9 with RedHat kernel 2.4.20-20.9smp. From the table, it is clear that the type inference times without the heuristics are impractical often exceeding 12 hours. However the algorithm with heuristics executes in just a few seconds, thus making it usable for models seen in practice.

7. CONCLUSION

In this paper, we presented a new type inference algorithm, including heuristics to reduce run-time, for a new structural type system that supports parametric polymorphism and a disjunctive type that allows component overloading. We show that, despite the NP-completeness of the inference problem, our algorithm for type inference has reasonable run-times (< 10 seconds) for systems seen in practice. These contributions permit low-overhead use of parametric polymorphism and component overloading, increasing component reuse in practice. This reduces model development times, increases the number of designs that can be explored, and results in better overall design quality.

Model Name	Explicit Type Instantiations w/o Type Infer.	Explicit Type Instantiations w/ Type Infer.	Basic Algorithm Run Time	Run Time with Heuristics
A	115	8	> 12 h	6.54s
B	116	8	> 12 h	6.58s
C	38	30	14.9s	0.12s
D	162	71	> 12 h	1.78s
E	147	63	> 12 h	4.72s
F	101	38	> 12 h	2.76s

A	A Tomasulo Style machine for the DLX instruction set.
B	Same as A, but with a single issue window.
C	A model equivalent to the SimpleScalar simulator [1].
D	An out-of-order IA-64 processor core.
E	Two of the cores from D sharing a cache hierarchy.
F	A validated Itanium II model.

Table 2: Experimental Results.

8. REFERENCES

[1] Burger, D., and Austin, T. M. The SimpleScalar tool set version 2.0. Tech. Rep. 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[2] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[3] Doucet, F., Otsuka, M., Shukla, S., and Gupta, R. An environment for dynamic component composition for efficient co-design. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2002).

[4] Doucet, F., Shukla, S., and Gupta, R. Typing abstractions and management in a component framework. In *Proceedings of Asia and South Pacific Design Automation Conference* (2003).

[5] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY, 1979.

[6] Harper, R. *Programming Languages: Theory and Practice*. Draft, 2002.

[7] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[8] Odersky, M., Wadler, P., and Wehr, M. A second look at overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1995), pp. 135–146.

[9] Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0*, 2001. http://www.systemc.org.

[10] Paterson, M. S., and Wegman, M. N. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing* (1976), ACM Press, pp. 181–186.

[11] Putzke-Roming, W., Radetzki, M., and Nebel, W. Objective VHDL: Hardware reuse by means of object oriented modeling, 1998. http://eis.informatik.uni-oldneburg.de/research/request.html.

[12] Seidl, H. Haskell overloading is dexptime-complete. *Information Processing Letters 52*, 2 (1994), 57–60.

[13] Smith, G. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming 23*, 2-3 (1994), 197–226.

[14] Vachharajani, M., Vachharajani, N., Penry, D. A., Blome, J. A., and August, D. I. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture* (November 2002), pp. 271–282.

[15] Xiong, Y. *An Extensible Type System for Component Based Design*. PhD thesis, Electrical Engineering and Computer Sciences, University of California Berkeley, 2002.