Benefits of Type Inference for an Object-Oriented Real-Time Language

Jan Gustafsson  Kjell Post  Jukka Mäki-Turja  Ellus Brorsson*
Department of Computer Engineering
University of Mälardalen
PO Box 883, S–721 73 Västerås, Sweden
E-mail: \{jgn, kpt, jma\}@mdh.se

Abstract

RealTimeTalk is a framework and an object-oriented language for distributed hard real-time systems. This paper presents past, present, and future work in the RealTimeTalk project. It concentrates on the language and programming part of RealTimeTalk, and the problems a dynamically typed language impose on a real-time system.

We look at how some of these problems can be solved by type inference and how type information can assist the compiler in accepting larger classes of real-time programs, generate more efficient and deterministic code, provide better estimation of execution times, and avoid run-time type errors. With type inference, one can have the best of two worlds — efficiency and type safety, without the need for manual type declarations. An example is used to illustrate these ideas.

1 Introduction & Background

As real-time applications become increasingly larger and more complex, demand for languages that makes real-time systems easy to develop, maintain, and extend will grow. This paper presents RealTimeTalk [BEG92, EGGB93] which is a framework and an object-oriented language for distributed hard real-time systems. Hard real-time systems are systems where all deadlines must be fulfilled.

As the name suggest, RealTimeTalk (RTT) is influenced by the Smalltalk system and its simplicity [GR89]. Smalltalk offers all the advantages of object-oriented languages: reusability, frameworks, rapid prototyping, incremental development, modularity, and extensibility. However, current Smalltalk implementations are not suitable for hard real-time systems because of some non-deterministic behavior, primary examples being garbage collection and method lookup schemes. In RTT, garbage collection and method lookup have been implemented in a deterministic manner to guarantee the temporal behaviour of the application [BEG92]. In addition, unbounded loops and data structures, dynamic class creation, and recursion is not allowed.

The main goal of RTT is to simplify the construction of predictable real-time systems. Models and designs are expressed either graphically or textually using the design objects in RTT [EHMS95].

In RTT we have separated the functional and temporal behaviour of the system. The functional requirements are checked through prototyping and testing. The temporal constraints are verified statically by the compiler and the WCET (Worst Case Execution Time) calculation tool [Gus94a, Gus94b] and are guaranteed at run-time by static scheduling.

2 Current Status

The RTT research group at the University of Mälardalen in Västerås and the Royal Institute of Technology (KTH) in Stockholm consists of seven people working on different parts of the RTT real-time language, tools and environment [EGG+95].

2.1 Summary of Contributions

At this stage, prototypes in the following two areas have been built:

- The RTT compiler and development tools;
- The RTT run-time system.

Evaluation of hardware architectures, suitable for the RTT system, is going on in parallel.
The RTT run-time system supports execution of RTT programs in a distributed target architecture. The distribution concept includes clock synchronization over the network (currently the CAN bus is supported [Tha95]). The RTT run-time system kernel is closely integrated with the off-line scheduler to achieve timeliness in the execution [ELL95]. The new version of the RTT operating system will support concurrent execution of tasks with both hard and soft real-time requirements.

The RTT development tools are shown in fig. 1. These tools have been developed with the use of Smalltalk/V, the compiler generator tool Bison, and C. Together, they run under Microsoft Windows.

The application editor is a graphical tool used to define the RTT design objects. The implementation of classes and methods is made textually in the RTT browser.

The RTT compiler takes the implemented classes and methods as input and generates C-code. The WCET tool (timing analysis) calculates the execution time and feeds the results back to the application editor. The total code and timing information is then passed to the RTT scheduler, which tries to find a feasible schedule for the system. If such a schedule is found, the C-compiler and linker finally generate the load module. If not, the system designer has to reconsider the design and implementation of the system.

In the text to follow, we focus on analysis tools to assist the compiler in accepting larger classes of programs and generate safer and more efficient applications.

### 2.2 Problem statement

Like Smalltalk, the current version of RTT is dynamically typed, i.e., types are associated with values rather than variables. This is in contrast to static typing where the type is associated with the variable. Both type systems have their merits [Bud91, page 107].

Dynamic typing is generally considered more flexible than static typing and also relieves the programmer from having to declare the types of variables. On the other hand, type errors often go undetected until catastrophic errors occur during execution. For example, Smalltalk happily accepts the construct `1+"hello"` until, when at run-time, the method `+` in class `Int` discovers that the type of its argument is not `Int`, leading to a run-time error.

When static types are known to the compiler, type conflicts can be caught at compile time. This is clearly superior to finding type errors at run-time which may require a lot of test cases and insight on behalf of the programmer. With type information, the compiler can produce more efficient code and provide better estimation of WCET for ambiguous operations, e.g., use of polymorphism.

Another source of run-time error in an object-oriented language is “method not understood”. This happens when the receiving object does not implement a method for the message. Like the abovementioned type error, this can be disastrous in a real-time system. One ad hoc way of dealing with this problem is to systematically rename methods so that it is always clear by looking at the program text which method will be invoked for a particular message. This, however, is against the very idea of polymorphism.

Another alternative is to obtain information about which class the receiver of the message is an instance of. This can be arranged by letting the programmer provide type declarations, like in C++ [Str91] or Ada [Ame83]. However, explicit declarations are a burden to the programmer, especially during prototyping. It would be more convenient if the compiler could figure this out by itself.

**Definition 2.1**

- **Run-time error**: Either caused by a type error or an exception.
- **Type error**: Not being able to find a method for the message, or trying to apply a primitive operation (like `+`) to improper arguments.
- **Exception**: Other events such as arithmetic overflow, index out bounds, etc.
3 The Future

We are currently in the process of designing a type inference system for RTT. Type inference was popularized by Standard ML [MTH90] and has subsequently been incorporated into other programming languages as well.

3.1 Benefits of Type Inference

The purpose of the type inference system for RTT is to annotate each variable occurrence with a type.

Definition 3.1 A type is a finite set of class names \{C_1, \ldots, C_k\} and represents the classes that a variable occurrence can be an instance of at run-time.

With type information at hand, the compiler can

- Reduce over-estimations of WCET calculations.
  When a receiver is known to be an instance of a set of classes, the WCET calculation is limited to these classes.

- Guarantee statically type-safe programs.
  Once types are inferred, the compiler can verify statically that all messages will be understood at run-time and that arguments to system builtins have the proper type. Of course, the compiler may be overly pessimistic and reject a program that would avoid an offending construct at execution.

- Eliminate run-time type checking.
  When programs are type-safe, there is no need for builtin primitives to check the type of arguments before using them. This will reduce run-time overhead.

- Produce more efficient code.
  When a receiver is known to be an instance of one class only, method lookup can be replaced with a function call. Efficiency can sometimes also be improved even if the receiver could be an instance of more than one class (cf. sect. 3.4).

- Characterize polymorphic recursion.
  In object-oriented languages, recursion has a deeper meaning, compared to other languages, since a function (a method) is identified not only by the name of the function, but also by the receiving object. Suppose an instance \textit{I} sends a message \textit{m}. Then we can differentiate between three kinds of recursion, in ascending order of generality:

- \textbf{Instance recursion}: when the execution of \textit{m} invokes the method for \textit{m} in \textit{I} again (perhaps transitively).

- \textbf{Class recursion}: when the execution of \textit{m} invokes a method for \textit{m} in an instance of the same class as \textit{I}.

- \textbf{Polymorphic recursion}: when the execution of \textit{m} invokes a method for \textit{m} in some other instance (perhaps \textit{I}). Thus, polymorphic recursion may not be recursive at all!

In RTT, recursion is not allowed because of the problem to determine the recursive depth of a data dependent recursion [Gus94a]. With type information, polymorphic recursion can sometimes be classified as non-recursive and therefore be allowed in RTT.

Example 3.1 Consider an instance \textit{I} of class \textit{C} executing \textit{m}, using the following method:

\begin{verbatim}
m ...
\end{verbatim}

\begin{verbatim}
  \textit{x} \textit{m}.
\end{verbatim}

\begin{verbatim}
  ...
\end{verbatim}

If the type of \textit{x} includes \textit{C}, the compiler must reject the polymorphic recursion, because it \textit{may} lead to (class) recursion.

In addition, type inference frees the programmer from having to declare types and avoids the problem of having to deal with incorrect declarations. Instead, the programmer can focus on the design problem. The benefits of type inference in a rapid prototyping environment have been observed by Palsberg and Schwartzbach [PS94, page 72]; they suggest that Smalltalk is used “typeless” in the design phase. Later, when the product is ready to run, type inference is used to detect type errors and optimize the execution (fig. 2).
### 3.2 Principles of Type Inference

A simple and intuitive way to infer types is to apply the techniques of abstract interpretation [CC77]. Given a program, the idea is to construct an abstract program which operates on *type* objects rather than the usual objects.

Consider the simple program in fig. 3. The corresponding abstract program, shown to the right, works with the types of $s$ and $x$, denoted $s#$ and $x#$, respectively.

Rather than representing types as simple data structures we promote types to full objects and obtain a surprisingly simple and intuitive correspondence between classes, objects, and messages in the source program, and type classes, type objects, and type messages in the abstract program. For example, in the abstract program, `new` creates a new type object. Messages to $s$ correspond to messages to $s#$ in the abstract program, where arguments have been replaced with their types.

Fig. 4 shows the types of $s$ and $x$ after each program statement. Variables have been subscripted after the statements in which they occur. As can be seen, the type of $x$ is $\{\text{Int, String}\}$ after line 5. Here, the compiler can detect that $x + 1$ in line 6 may lead to “message not understood”, should $x$ happen to be a `String`.

This simple idea can be extended to handle selection and iteration as follows. For selection, compute the types in both branches and merge the information at the join point. For iteration, repeatedly re-compute and add type information for all variable occurrences in the body of the loop until the information has stabilized. Termination is guaranteed by the monotonicity of the union operation and the finiteness of type representations and the number of type constructs in the abstract program’s loop.

Corresponding to the “built-in” classes of the system there exists type-classes (like `Set#`) with methods to emulate the type behavior of the original class. For example, in `Set#` we have the following methods:

- `add: obj#`  
  `contents#: contents# \cup \{obj#\}`
- `at: i#`  
  `^\{\text{contents#}\}`

`contents#` represents the element type in `Set#`.) The programmer is currently not allowed to redefine built-in classes like `Set`.

### 3.3 Example

The following example serves to demonstrate the ideas presented in this paper. For simplicity, the real-time constraints of the system are not defined; it is our hope that it is clear how our reasoning may be generalized to a “real” real-time system.

**Example 3.2** A robot picks up fruit from a conveyor belt (fig. 5). Fruit heavy enough to comply with EC requirements are put in a container box, all other fruit is discarded. When the weight of the box exceeds a certain limit, the box is pushed aside.

The system is organized into the following classes:

- **ConveyorBelt**: Carries a collection of fruit.
- **Robot**: Picks a piece of fruit from the conveyor belt and drops it in the box, if it is heavy enough, otherwise it is discarded.
- **Box**: Stores a collection of fruit. Its weight is the sum of its contents.
- **Pusher**: Pushes the box aside when it is full.

The code for the system is given in the appendix. As an example of output from the type inference procedure, the type representation of `theRobot` after line 74 is shown in fig. 6.
Figure 4: Execution trace showing the types of variable occurrences $s_i$ and $x_i$ after line $i$ has been executed.

Figure 6: Inferred type for theRobot after line 74.

3.4 Discussion

The benefits of type inference for example 3.2 is given below. It is the polymorphic method weight that is the focus of our discussion.

- Reduce over-estimations of WCET calculations.

The inference algorithm can figure out that in line 30, 31, and 43, the type of the receiver for the weight message is $\{\text{Orange, Banana}\}$. In line 8, the receiver's type is $\{\text{Box}\}$. As all three classes implement a weight method, the message weight will always be understood.

Without type information, the compiler must calculate WCET for the weight message as the maximum of the execution times for the weight method in classes Orange, Banana, and Box. The execution time for weight in Box will certainly exceed the execution time for weight in Orange and Banana (because it invokes them). Thus, without type information, the WCET for the expressions in line 30, 31, and 43 will be overestimated. With type information, the WCET is correctly estimated.

- Guarantee statically type-safe programs.

Recall that in line 30, 31, and 43, the receiver for weight has the type $\{\text{Orange, Banana}\}$ whereas in line 8, the receiver’s type is $\{\text{Box}\}$. As all three classes implement a weight method, the message weight will always be understood.

Let us examine the safety benefits of the type inference system. Suppose the programmer wants to add vegetables as a separate class to the system but forgets to write the weight method. If vegetables are put on the conveyor belt, the compiler would be able to spot the error.

- Eliminate run-time type checking.

The compiler is able to deduce that all invocations of system builtins, e.g., primitives for $+$ and $>$, have arguments of type Int so that run-time checks are unnecessary.

- Produce more efficient code.

1The WCET for weight in Box depends on the size of the contents of Box. In a real implementation, there must be an upper bound on this size.
Recall that in line 8, the receiver can only be an instance of Box. Since the receiver is known, method lookup can be replaced with an ordinary function call.

In the more general case, when the receiver’s type is \( \{ C_1, \ldots, C_k \} \), \( k > 1 \), execution time can still be improved, at the cost of code expansion. Recall that we were able to deduce that only instances of Orange and Banana could appear on the conveyor belt. The compiler can use this information to split the body of the method moveObjects into two cases, one for each class in anObject’s type, preceded by a class test of anObject to determine which branch to go (fig. 7). In each branch, method lookups can be replaced with function calls.

- Characterize polymorphic recursion.

In the example we find a case of polymorphic recursion in line 43. This expression is really not recursive at all, since Box’s method weight only calls the weight method in Orange and Banana, not leading to any more weight messages. This code is thus fully acceptable in RTT.

### 4 Related Work

Various type systems for object-oriented languages have been investigated before. We briefly mention some results in the Smalltalk community.

Palsberg and Schwartzbach [PS94] describes a type inference procedure for Smalltalk. We have adopted their view of a type, i.e., as a set of classes. Their work could be used in the RTT project if it could be extended to handle blocks. It would also be useful if variable occurrences, rather than just variables, were annotated, giving a more precise estimation. For example, in

\[
x := 7.
\]
\[
x := \text{true}.
\]
\[
x \text{isTrue}.
\]

we would like \( x \) to have the type \{Int\} after the first assignment, and \{Bool\} after the second.

Graver [Gra89] presents a version of Smalltalk called “Typed Smalltalk” where the compiler infers types for some variables whereas others must be explicitly given by the programmer. Graver’s work essentially supersedes earlier work done by Suzuki [Suz81].

Barnard [Bar93] presents a strategy for inferring types, using dataflow analysis. Again, this could be useful, if implemented, but one should adapt it for real-time systems, where the entry point for the application is known.

### 5 Conclusions

We began by presenting past and ongoing research in the RTT project, which is concerned with simplifying the construction of predictable real-time systems, and then focused on the language and the problems that arise when using a dynamically typed object-oriented language for real-time systems.

The remainder of the paper pointed out how type inference for these languages can benefit the development of software for real-time systems. Specifically, type information leads to better estimation of execution times, more efficient and deterministic code, no run-time type errors, and the possibility to recognize polymorphic recursion as non-recursive.

Future work involves an implementation of the type inference system described in section 3.2.

### 6 Acknowledgements

This work was supported by The National Board for Industrial and Technical Development (Proj. no 93–3180), Västmanlands County Administrative Board and fundings at the University of Mälardalen and the Royal Institute of Technology.
References


Appendix: Example Code

Class: Pusher
Inherits: Object
Instance variables: theBox

```
connectTo: currentBox
"Connect the pusher"
theBox := currentBox

newObject
"Tell receiver a new object has arrived"

| boxWeight |
boxWeight := theBox weight.

Transcript show: 'Box weight = ',
boxWeight printString, '.

(boxWeight > 150)
ifTrue: [Transcript show: ', will push'; cr]
ifFalse: [Transcript show: ', will not push'; cr]
```

Class: Robot
Inherits: Object
Instance variables: theConveyorBelt theBox thePusher

```
connectTo: aConveyorBelt and: aBox and: aPusher
"Connect the robot"
theConveyorBelt := aConveyorBelt.
theBox := aBox.
thePusher := aPusher

moveObjects
"Move objects to the box if they are big enough"

| anObject |
[theConveyorBelt notEmpty] whileTrue: [
anObject := theConveyorBelt removeFirst.
Transcript cr;
show: ((anObject class) printString),
' weight = '; 
show: ((anObject weight) printString); cr.
(anObject weight > 40)
ifTrue: [
Transcript show: 'Put it in the box'; cr.
theBox add: anObject.
thePusher newObject]
ifFalse: [
Transcript show: 'Throw it away'; cr]]
```

Class: Box
Inherits: OrderedCollection
Instance variables:

```
weight
"Calculate the total weight of contents"

| acc i |
acc := 0. i := 1.

[i ≤ (self size)] whileTrue: [
acc := acc + (self at: i) weight.
i := i + 1].
```

Class: Fruit
Inherits: Object
Instance variables: weight

```
setWeight: newweight
"Set the weight"
weight := newweight

weight
"Answer the weight"
```

Class: Orange
Inherits: Fruit
Instance variables:

Class: Banana
Inherits: Fruit
Instance variables:
<table>
<thead>
<tr>
<th>Transcript code</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
</tr>
<tr>
<td>54</td>
</tr>
<tr>
<td>55</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>57</td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td>59</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>61</td>
</tr>
<tr>
<td>62</td>
</tr>
<tr>
<td>63</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>65</td>
</tr>
<tr>
<td>66</td>
</tr>
<tr>
<td>67</td>
</tr>
<tr>
<td>68</td>
</tr>
<tr>
<td>69</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>71</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>73</td>
</tr>
<tr>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange weight = 50</td>
</tr>
<tr>
<td>Put it in the box</td>
</tr>
<tr>
<td>Box weight = 50, will not push</td>
</tr>
</tbody>
</table>

| Banana weight = 70 |
| Put it in the box |
| Box weight = 120, will not push |

| Orange weight = 30 |
| Throw it away |

| Banana weight = 60 |
| Put it in the box |
| Box weight = 180, will push |