# ART Memo 27 (Version 0)
# The Semantics of an Augmented Redex Trail

Olaf Chitil

January 14, 2002

## 1 Introduction

Discussion of the meaning of a trace, an Augmented Redex Trail, in practise.

## 2 Functions and Applications with Arity

### 2.1 Functions with Arity

In pure $\lambda$-calculus any application of a $\lambda$-abstraction is a redex. In contrast, in Haskell a function definition usually has several arguments and it would be rather unnatural to model Haskell functions as taking one argument at a time. Hence every Haskell function (a defined variable or a $\lambda$-abstraction) has an arity. An application of a function of arity $n$ to $m$ expressions is a value iff $n > m$.

### 2.2 Applications with Arity

The presence of n-ary functions does not force the presence of $n$-ary applications. In fact, given a language with n-ary functions, it is usually easier to formulate an operational semantics with binary applications than with n-ary applications. A semantics for the latter requires the distinction between three cases when considering an $m$-ary application of an $n$-ary function:

1. $n > m$: the application is undersaturated; the application is a partial application, it is a value.

2. $n = m$: the application is saturated, it is a redex.

3. $n < m$: the application is oversaturated; *part* of the application is a redex.

The semantics of an oversaturated application is the most complex of the three and just that case does not exist if all applications are binary.

Saturatedness of an application is only defined for an application whose function subexpression is a value (only functional *values* have an arity). Hence in a higher-order language it is not possible to classify applications in the *program* as undersaturated, saturated or oversaturated. Only when an application in a computation is evaluated its functional part is evaluated and then the saturatedness is known.

However, the trace language ART has n-ary applications, because it would be rather expensive in time and space to build every n-ary application from binary applications. So an n-ary application in the program will give rise to n-ary applications in the trace.

## 2.3 Representation in the Trace

An application node is created in the trace at the moment when a redex is reduced and its rhs has to be created in the trace. At that point the saturatedness of the applications is in general still undetermined.

Different sorts of applications are represented as follows in the trace. During computation and hence trace creation the sort of an application can still change. I hope this is correct. The square brackets refer to the planned new trace design.

**Unevaluated** Saturatedness has not been determined.

A SatA pointing to the application node just in front of it. [An application node with result root.]

**Entered but not evaluated** Saturatedness may have been determined.

A SatB pointing to the application node just in front of it. [An application node with result pointing to rhs as in following cases?]

**Evaluated, undersaturated** A SatC pointing to the application node just in front of it. [An application node with result pointing to itself.]

**Evaluated, saturated** A SatC with the application node (redex) just in front, but it points to the rhs. [An application node with result pointing to the rhs.]

**Evaluated, oversaturated** A SatC. The application node is just in front. The SatC points to another Sat(A/B/C) with an application node in front. This second application can again be of any of the sorts described here. Both application nodes have the same parent (this marks oversaturatedness). The second application node applies the result of the redex of the first application to the superfluous arguments of the first application. [The result pointer takes the role of the Sat and it points to the second application, not just its result pointer.]

Example:

```
g z = z
f x y = g
main = f 3 2 1
```

$$\texttt{main} \rightarrow \texttt{f 3 2 1} \rightarrow \texttt{g 1} \rightarrow \texttt{1}$$

Both `f 3 2 1` and `g 1` have `main` as parent. `g` has `f 3 2 1` as parent.

So an application node in the trace is either a redex or a value, except if it is an oversaturated application. It is the task of the viewing tools to recognise oversaturated applications and to rearrange that part of the trace to show it correctly (currently not done).

Here `f 3 2 1` and `g 1` have to be replaced by an application of the reduction `f 3 2` $\rightarrow$ `g` to the single argument `1`.

It seems impossible to create the trace for oversaturated applications correctly. At the latest when an application needs to be evaluated, it must be recorded in the trace. However, only when the function part is evaluated the saturatedness of the application is known. If the computation is aborted between these two events, this must be visible in the trace. Then the result of the reduction is $\perp$ and the aborted computation is reachable in the trace though the function part.

# 3 Needed Values

Freja shows values (in arguments or results) only if they are needed by the computation. Hood even only shows values which are needed by the context its observation.

An ART contains and the Hat tools show values which are never needed by the computation. Consider for example:

```
f x = [x,x,x]
main = print (null (f 42))
```

The trace contains the expression `[42,42,42]` and the tools show this expression for example as argument of `null`, although in the whole computation only the outermost cons constructor of the list is ever needed.

An ART contains such values, because every time a redex is reduced the whole rhs is created in the trace. The tools show such unneeded values, because currently it is impossible to determine from the trace if a value was needed or not; at least for literals and saturated constructor applications, for partial applications neededness can be determined.

Not including unneeded values would yield smaller traces but might be more complicated and increase tracing time. Marking needed values is straightforward but will increase trace size and tracing time slightly (let values have Sat's or result pointers like applications).

# 4 Unevaluated Applications

A trace contains unevaluated applications. These unevaluated applications are subexpressions of applications which were evaluated by the computation (or were about to be evaluated when the computation was aborted). Both hat-detect and hat-observe usually do not show these unevaluated applications but represent them by a symbol `?`. Unfortunately hat-trail is not able to show unevaluated applications as `?`, so that speculations that this might improve the usability of hat-trail could not be verified. Knowledge about unevaluated applications is certainly not needed to answer the questions of hat-detect and avoiding superfluous details improves the usability of a viewer.

A trace without unevaluated applications would be smaller, but it is unclear if creating it would be easier. Current experience with implementing untraced code suggests that it could even be more complicated and time consuming.

# 5 Untraced Computations

Part of a program can be designated as untraced to first of all avoid unnecessary detail in the trace. A good first intuition is to view untraced functions as internal primitives; for untraced functions only inputs and outputs are visible. However, the fact that traced or untraced functions can be part of inputs or outputs makes a definition of untracedness hard.

## 5.1 Properties of the Trace of Untraced Computation

A few desirable properties:

- the application of an untraced function called from traced code is in the trace, including (needed) arguments
- the (needed) value of the redex is in the trace
- there is no distinction in input and output between first-order and higher-order values (partial applications); all are in the trace
- if evaluation of the untraced function causes evaluation of a traced function, the whole computation of this traced function, from application to final value, are in the trace
- the trace of an unevaluated untraced application is the hidden original application of the untraced function
- the parents of all traces of untraced code are the hidden original application of the untraced function (currently the top of the rhs of the original application has the original application as parent, unhidden)
- intermediate untraced computation may not need SatB's; important are only SatB's in the result to represent $\perp$
- untraced code that does not appear in any redex or result mentioned above does not yield any trace

An application belongs into the trace if the application or the function value are traced (and it is needed?). The program statically determines the tracedness of an application, but the tracedness of the function part is only known after it has been evaluated.

## 5.2 Visibility of Identifiers

It seems natural to demand that when an abstract data type is designated as untraced, the trace should only show those identifiers of the abstract data type which are in the public interface. The internal representation of abstract data and probably also partial applications of internal functions should not be visible (it might be desirable to see the mapping a la Hood of such a partial application, but not the function name and the supplied arguments).

To implement such kind of tracing, the visibility frontier needs to be *defined*. There exist several interfaces between traced and untraced code, because entities of a single untraced module may be imported from several traced modules via different routes with different visibility. Second, the visibility of every identifier must be accessible at program transformation or at tracing time.