Programming Research Group

INTENTIONAL PROGRAMMING:
A HOST OF LANGUAGE FEATURES

Eric Van Wyk
Oege de Moor
Ganesh Sittampalam
Ivan Sanabria Piretti
Kevin Backhouse
Paul Kwiatkowski

PRG-RR-01-21

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

1 Introduction

Programming languages and programming tasks are rarely a perfect fit: often a program could be much clarified by using a number of tailored language features, but the cost of introducing those features in the language is perceived as too high. If a programming language could be implemented in a highly modular fashion, that cost might be lower. To achieve such modularisation is the goal of *Intentional Programming*[1]; language features are called *intentions* to emphasise the fact that language features can be tailored to the programmer's wishes. One starts with a *host language*, that is subsequently extended by adding new domain-specific features. Crucially, these features should be *composable* wherever possible; independent language features should co-exist peacefully without each needing to be explicitly aware of presence or absence of others. Indeed, given a sufficiently broad library of intentions, it should be possible to construct a new language from the ground up.

When adding new intentions to a language it is essential that they be implemented in an efficient manner. It is critical that programs developed via Intentional Programming are not any less efficient than those developed by traditional means. Our aims as intention developers are to build intentions which are both expressive and efficient.

Aims of this paper This paper is a report to our colleagues in the Intentional Programming team at Microsoft about a model and prototype implementation of *intentional programming* whose design owes much to their R5.0 implementation of *reduction*. (Intentions can often be described in terms of constructs in the host language or other lower-level intentions. For this reason, the modular compilation of intentions is referred to as *reduction* since higher level intentions are said to "reduce" to lower level ones.) Its primary aim is to explain and formalise reduction in the simplest possible terms, namely as a program in the lazy functional language Haskell. That program thus serves a dual purpose: it provides a formal semantics of the main concepts of reduction, and it also provides a platform for low-cost experiments.

A secondary aim of this paper is to relate the ideas of intentional programming to work of others, in particular in the literature on attribute grammars. Given the close connection to attribute grammars, one might expect that we explain intentional programming entirely in those terms, as a particular attribute grammar system with some particular features to achieve modularity. This would however be overly restrictive: in writing this paper, we want to promote the ideas themselves, not a particular implementation. Also, we feel that it is important to experiment with variations of these ideas. Therefore we shall present intentional programming as a particular *style* of writing a lazy functional program, providing maximum flexibility in experimentation. A disadvantage of this approach is that we have to dispense with certain static checks that a attribute grammar preprocessor would perform as a matter of course. We do however get the advantage of the type system of Haskell, which helps to give crisp definitions of various concepts.

[1]Intentional Programming [Sim96, Sim99] is the brainchild of the founder of Microsoft's applications division Charles Simonyi.

The choice of a lazy functional programming language for our model is an obvious one. One of the main concepts of R5.0 is that of a 'question'. This is equivalent to an *attribute* in attribute grammar terminology; the name used in R5.0 comes about because they are evaluated in a demand-driven fashion. In the remainder of this paper, we shall stick to the standard terminology.

By formulating a precise semantics of reduction, we have been able to solve two outstanding problems in R5.0. In particular, it was unclear how to deal with transformations that copy subtrees: our model obviates the difficulty, by representing trees as functions whose parameters are provided by the context. This formulation also simplifies many of the other complexities associated with the connections between tree nodes.

Another problem concerned the meaning of cyclic attribute dependencies, which required a feature called *rollback* in the original design. The intended semantics of rollback was however far from clear. Here our answer is a fix-point-finding evaluation mechanism. Again, this solution is a natural consequence of our formal definitions.

For simplicity and ease of exposition, our model omits some of the more advanced features of R5.0 (such as system support for "build targets" and error handling). Including them would not pose any conceptual difficulties; we shall discuss these and other "high-level" features after we have given a detailed account of the model.

Relation to attribute grammars Readers who wish to understand the connections with the literature on attribute grammars may find it useful to have a clear understanding at the outset of the ways in which our model of reduction improves over the pure attribute grammar paradigm. Below we briefly list these improvements; each point will be explained at leisure later on in the paper.

- Trees are regarded as functions of their context. This viewpoint enables the programmer to refer to non-local parts of the tree in a semantically transparent way. The importance of this feature was first put forward in the *reference attribute grammars* of Hedin [Hed99]. Our solution is, however, more akin to that of the *higher-order attribute grammars* of Swierstra *et al.* [SV91].

- A non-terminal need not define all its attributes explicitly: some attribute values may be inherited from another node in the tree. We call this phenomenon *forwarding*. The idea was first put forward by Paul Kwiatkowski of Simonyi's team at Microsoft. Similar (but less powerful) features can be found in the Lisa system of Mernik [MLAV99].

- The attribute grammar can furthermore be sliced into *aspects*. An aspect is a module where the definitions of related attributes are grouped together. Such an aspect could involve the implementation of a new programming language feature such as *mkArray* (described below), or merely the definition of some useful analysis information over existing intentions, such as the absence of side-effects. This idea is a natural application of Kiczales' *aspect-oriented programming* [KLM$^+$97] to the problem domain of language implementation.

- Program analyses are often conveniently specified as *circular* attribute grammars. Formally, we can think of an attribute grammar as a mapping from trees to sets of equations over the attribute values. If the attribute dependencies are non-circular, the equations have a unique solution; if a cycle exists, we wish to compute the *least* solution with respect to an appropriate partial order. As said, our interest in this idea was inspired by *rollback* in R5.0. The observation that circular attribute grammars are a powerful tool in program analysis is due to Rodney Farrow [Far86].

In [VWdMBK02], we show how forwarding and production valued attributes can be added to higher order attribute grammars as they are traditionally presented. The model described here more closely follows the R5.0 design and implementation and can be seen as an embedding of that model into Haskell in much the same fashion as Johnsson [Joh87] shows how attribute grammars can be embedded into lazy functional programming languages.

Motivating example For concreteness, let us consider the problem domain of image synthesis and manipulation. Arrays that model bitmaps play an important role in this domain; consider the following program to allocate and initialise a gradient-filled bitmap:

```
let $v$ := vec ($x\_size$), $i$ := 0 , $j$ := 0
in while ($i < x\_size$) do
     $v[i]$ := vec ($y\_size$) ;
     $j$ := 0 ;
     while ($j < y\_size$) do
       $v[i][j]$ := $i + j$ ;
       $j$ := $j + 1$
     end;
     $i$ := $i + 1$ ;
   end
end
```

One convenient shorthand for this would be the following:

$$mkArray\ (x\_size,\ \lambda\, i\ \rightarrow\ mkArray\ (y\_size,\ \lambda\, j\ \rightarrow\ i + j))$$

In keeping with standard notation, the symbol λ is used to introduce the name of a parameter for an anonymous function to the body that follows the symbol $\rightarrow$. More generally, the *mkArray* intention is used as follows:

$$mkArray\ (size,\ \lambda\, i\ \rightarrow\ E)$$

This expression creates an array of given *size*, the second argument being an anonymous function that indicates what value should be stored at position i in the array. In principle the *mkArray* expression is merely shorthand, namely for

```
let $v$ := **vec** ($size$), $i$ := 0
in while ($i < size$) do
        $v[i] := E$ ;
        $i := i + 1$
    end;
    $v$
end
```

If this was all we wanted, *mkArray* could be introduced as a procedure in the host language. However, the benefit would be greater if we optimised occurrences of *mkArray*, lifting loop invariant code and caching inner expressions in nested loops. For example, we would hope that a nested use of *mkArray* such as

$$mkArray\ (size_1,\ \lambda\, i\ \rightarrow\ (mkArray\ (size_2,\ \lambda\, j\ \rightarrow\ f(i, 2 * i)\ +\ g(j))))$$

is first optimised to

```
let $x := mkArray\ (size_2,\ \lambda\, j \rightarrow g(j))$
in  $mkArray\ (size_1,\ \lambda\, i \rightarrow$ let $y := f(i, 2 * i)$
                                  in $mkArray\ (size_2,\ \lambda\, j \rightarrow y + x[j])$
                                  end )
end
```

before applying the expansion above. To see why this is desirable, consider the original expression: it requires $size_1 * size_2$ evaluations of f and g. By contrast, the optimised version requires $size_1$ evaluations of f and $size_2$ evaluations of g. Naturally these transformations need a number of conditions, in particular that f and g are free of side-effects. This, in turn, requires that we have some knowledge of the host language, in which the new feature is embedded. Such assumptions should be recorded as an *interface* of the new feature, so that it can be regarded as a module in the language description.

To appreciate how delicate such transformations can be, observe that lifting loop invariant code is not valid in the following instance. The array expression

$$mkArray\ (size,\ \lambda\, i\ \rightarrow\ \textbf{if}\, false\ \textbf{then}\ 1/0\ \textbf{else}\ i)$$

is *not* semantically equivalent to

```
let $x := 1/0$
in  $mkArray\ (size,\ \lambda\, i \rightarrow$ if $false$ then $x$ else $i$)
```

So when introducing *mkArray*, the language designer must take the existence of conditional expressions into account. Naturally such assumptions could be problematic if we wish to add further features to the language. Consider, for example, the introduction of a new feature of *case* expressions, so that

$$\textbf{case}\ x\ \textbf{of}\ c_1\ \rightarrow\ e_1;\ c_2\ \rightarrow\ e_2;\ \ldots;\ c_n\ \rightarrow\ e_n;\ \textbf{otherwise}\ \rightarrow\ e$$

is equivalent to

```
let $t$ := $x$ in
  if $t = c_1$ then $e_1$
  else if $t = c_2$ then $e_2$
  else if ...
  else if $t = c_n$ then $e_n$
  else $e$
end
```

Do we now have to go back and modify the existing *mkArray* implementation? Clearly that would be undesirable, and as we shall see, it turns out to be unnecessary: *case* and *mkArray* can be added to the host language as independent components.

Overview Throughout, *mkArray* is used as a running example. We start in Section 2 with a brief overview of the host language. This overview will serve to introduce the view of trees-as-functions, forwarding, and aspects which are the building blocks of our intention definitions. Next in Section 3, we discuss in detail how the *mkArray* intention is added to the host language. In Section 4 we show how a *case* intention can be added in such a way that the original definition of *mkArray* remains unchanged. Finally, in Section 5 we discuss current status and future directions. Appendix A provides a few notes about the lazy functional language Haskell [PJHA$^+$99] used to define our prototype system.

2 The host language PLX

PLX is a simple imperative language with let bindings, lambda expressions (anonymous functions), closures, basic arithmetic, vectors, and garbage collection. Since the semantics of PLX is very similar to that of more complex languages such as ML, a brief informal description of its features will suffice.

PLX has a standard assignment statement and the control flow constructs **if** and **while**. The variable declarations and initialisations in a **let** binding are processed sequentially, as is the sequence of expressions that compose the body. There is no syntactic distinction between expressions and statements; some expressions, like assignment, have side–effects, some do not. A let binding evaluates to the value of the last expression in its body. Memory for vectors is allocated on the heap by the **vec** construct and is automatically de–allocated by the garbage collector.

2.1 The implementation of PLX as a set of intentions

All PLX language constructs are defined as *intentions.* An intention definition is similar to a production in attribute grammars in that it defines the intention's abstract syntax and a set of computations defining properties or attributes of an intention instance based on its context. An intention's *context* consists of its parent and children in the abstract syntax tree (AST) and, in the case of variable references, its declaration. Technically, links from a variable reference to its declaration cause the AST to be a graph instead of a tree but we will use the term "tree" regardless. Since there are several circumstances in which we use the phrases "tree", "node", "intention", and "intention definition" it is very helpful to provide a precise *type* for each of these concepts.

For simplicity, the context free grammar which defines the abstract syntax for PLX has only one non–terminal type, called *Node*, and thus nodes in the AST are said to have type *Node*. We shall think of such nodes as records where fields correspond to attributes and are thus seen as *finite maps* from attribute names to attribute values.

2.1.1 Intention definitions

In several cases we will work with trees which have only a partially specified context; that is, their parent and possibly children may not yet be specified. Trees which are to become subtrees of larger trees – that is, trees whose root has specified its children but has not specified its parent – are functions of the type

$$\textbf{type}\ \mathit{SemTree} = \mathit{Node} \rightarrow \mathit{Node}$$

These trees are functions which take a parent node as a parameter and use attribute values from this node to generate the root node of the sub tree. Typically, the information available from the parent includes the environment, which records all identifiers in the current scope. Indicating that the tree might require information from its parent by giving it this type mirrors the R5.0 notion of bidirectional tree links, which a child node uses to access its parent.

An *intention* is a function over a node's context[2] which includes child trees and a parent node that generates a new node: it corresponds to a production in attribute grammars. For example, the *while* loop intention, whose context is its parent, condition child and loop body child, is defined by a function with the type[3]

$$\mathit{while} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node}$$

This type declaration states that *while* has three parameters. The first two parameters are (the semantics of) the condition and body of this while loop. The type of these parameters is *Node* → *Node* (*SemTree*) since the child trees do not yet have parent nodes specified. The third parameter is the parent node of the while intention. The **while** intention can be implemented as follows:

$$\begin{aligned}
&\mathit{while}\ \mathit{cond}_f\ \mathit{body}_f\ \mathit{parent}_n = \mathit{while}_n\\
&\quad \textbf{where}\ \mathit{while}_n = \mathit{mkwhile}\ \mathit{cond}_n\ \mathit{body}_n\ \mathit{parent}_n\\
&\qquad\qquad\ \ \mathit{cond}_n = \mathit{cond}_f\ \mathit{while}_n\\
&\qquad\qquad\ \ \mathit{body}_n = \mathit{body}_f\ \mathit{while}_n
\end{aligned}$$

[2]For some intentions additional trees and nodes can be part of the context. As we will see a variable reference's context includes the variable's declaration node.

[3]The functional programming purist might write this type as the equivalent $\mathit{while} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{SemTree}$.

The node $while_n$ is generated by the function *mkwhile* from information, that is, attribute values, on the context nodes $cond_n$, $body_n$, and $parent_n$. The child nodes $cond_n$ and $body_n$ are constructed by their respective parameter functions $cond_f$ and $body_f$ which take as their parent the while node $while_n$. This appears to be a cyclic definition of $while_n$. Fortunately, demand-driven evaluation means that this computation is well-defined, as an evaluation order on (node,attribute) pairs exists.

As said, a node is viewed as a record whose fields are attribute values. The function *mkwhile* builds such a record as follows:

$$\begin{array}{l}
mkwhile :: Node \rightarrow Node \rightarrow Node \rightarrow Node \\
mkwhile\ cond_n\ body_n\ parent_n \\
\quad = mknode\ [\ \begin{array}{lcl} typei & = & nullType, \\ pp & = & \text{``}while\ (\text{''} +\!\!+\ cond_n.pp +\!\!+ \text{``})\ do\ \text{''} +\!\!+ body_n.pp +\!\!+ \text{``}\ end\text{''}, \\ env & = & parent_n.env, \\ jbc & = & mk\_while\_jbc\ (cond_n.jbc,\ body_n.jbc)] \end{array}
\end{array}$$

Here *mknode* is a record constructor function which when given the above field definition list builds a record with four fields, one defining each of the attributes *typei*, *pp*, *env* and *jbc*. The (.) operator is the record indexing function used to access attributes on the nodes in an intention's context. Above we have specified that the environment *env* of the while loop is the environment of its parent and that the Java byte code *jbc* of the while loop is a function of the Java byte code of its children. The pretty printing attribute *pp* is of type *String* and it is a text representation of the code represented by this intention. In most instances we will not write the field definitions in a separate function as we have above with the *mkwhile* function, but will "inline" them directly in the intention definition.

2.1.2 Record operations

To implement nodes as records, we define our own set of record operators since some features we require are not commonly provided in functional programming languages.

The function (.)[4] is a polymorphic function with type

$$(.)\ ::\ Node\ \rightarrow\ Attr\ \alpha\ \rightarrow\ \alpha$$

which takes a node, an attribute of type $Attr\ \alpha$ (α is a type variable) and returns a value of type α.

A record field definition takes the form $attribute\ =\ value$. We model such a field definition as a (infix) function of type $Node \rightarrow Node$ which, when applied to a node, generates a new node with all the same field values, except for field *attribute* which has the new *value*. The type of the field definition operator (=) is therefore

$$(=)\ ::\ Attr\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ Node\ \rightarrow\ Node$$

[4]In Haskell, (.) normally denotes function composition, but we choose instead to use this symbol for accessing fields in records.

Note that we are overloading the existing Haskell definition operator (=) but the context will indicate which operator we are using.

It remains to define the function *mknode* itself. It takes a list of field definitions, and it folds them, using function composition, into a single $Node \rightarrow Node$ function which it applies to an initial node, *initNode*. To wit, the definitions are[5]

$$
\begin{array}{l}
mknode :: [Node \rightarrow Node] \rightarrow Node \\
mknode\ attrDefs\ =\ initNode \oplus attrDefs \\
\\
(\oplus) :: Node \rightarrow [Node \rightarrow Node] \rightarrow Node \\
node \oplus [\,] = node \\
node \oplus (attrDef : attrDefs) = attrDef\ (node \oplus attrDefs)
\end{array}
$$

The auxiliary function $(\oplus)$ is simply a *field overriding* operator so that for $n = n' \oplus attrDefs$, the node n has the same field values as n' for all fields *except* those explicitly defined in *attrDefs*. This function is widely used in what follows. It appears in our definition of *copy rules*, *aspects*, as well as a possible implementation of a defaulting mechanism called *forwarding*, all described below.

2.1.3 Type intentions

The type of a PLX construct is defined by its *typei* attribute of its AST node. The language of PLX types is also represented as trees of intention *Node*s and the root node of such trees are stored in the *typei* attribute. These are very similar to trees in the PLX AST with the exception that a type intention's context does not include its parent, only its children (if it has any) which represent component types. It would be possible to have types that depend on their parent as well, but we do not need such a rich type system for our expository purposes. The decision that types should not depend on their parents is roughly equivalent to the us of unidirectional links within R5.0.

As an example of a simple type, consider the numeric type intention below:

$$
\begin{array}{llll}
numberType :: Node & & & \\
numberType = mknode & [\ isNumberType & = & True \\
 & \ \ isEqToType & = & (\lambda\ ti \rightarrow ti.isNumberType) \\
 & \ \ \ldots\,] & &
\end{array}
$$

The type attribute *isNumberType* is a Boolean value and it is defined to be true. As we shall see shortly, a default mechanism defines instances of *isNumberType* to be false for all other type intentions. The attribute *isEqToType* is a function $Node \rightarrow Bool$ that tests for type equality; it is defined by all type intentions. It should be noted that there is no type coercion or notion of subtype in PLX; A more sophisticated typing framework would not pose any conceptual challenges for our model.

To illustrate the use of *numberType*, here is a definition of numeric constants that specifies *typei*:

[5] Readers unfamiliar with Haskell may find it helpful to refer to the Appendix A

$$
\begin{aligned}
&\mathit{numericConst} :: \mathit{Terminal} \rightarrow \mathit{Node} \rightarrow \mathit{Node}\\
&\mathit{numericConst}\ \mathit{num}_n\ \mathit{parent}_n \ = \ \mathit{num\_const}_n\\
&\quad \textbf{where}\ \mathit{numconst}_n \ = \ \mathit{mknode}\ [\ \begin{aligned}[t] &\mathit{typei} &&= \mathit{numberType},\\ &\mathit{pp} &&= \mathit{num}_n.\mathit{lexeme},\\ &\mathit{jbc} &&= \ldots,\\ &\ldots\ ] \end{aligned}
\end{aligned}
$$

For simplicity the *Terminal* type can be considered to be equivalent to a *Node* which defines attributes relevant to terminal symbols such as *lexeme*. The same attribute lookup function (.) is used to reference these attributes. Since terminal symbols do not depend on their context, we do not have to instantiate the num_n node from a parameter num_f function as the num_n node is passed as a parameter to *numericConst*.

A slightly more interesting example is *sequential composition*, which takes two expressions, that may have side effects, and creates a single expression that evaluates the subexpressions in order. The value of the new expression is the value of the second expression after the first has been evaluated. The type of the new expression is the type of the second. This operator is very similar to the C language comma operator (,). Its definition is given below.

$$
\begin{aligned}
&\mathit{exprSeq} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node}\\
&\mathit{exprSeq}\ \mathit{expr}_f\ \mathit{expr}'_f\ \mathit{parent}_n \ = \ \mathit{this}_n\\
&\quad \textbf{where}\ \mathit{this}_n = \mathit{mknode}\ [\ \begin{aligned}[t] &\mathit{typei} &&= \mathit{expr}'_n.\mathit{typei},\\ &\mathit{pp} &&= \mathit{expr}_n.\mathit{pp} \mathbin{+\!\!+} \text{“; ”} \mathbin{+\!\!+} \mathit{expr}'_n.\mathit{pp},\\ &\mathit{jbc} &&= \ldots\ ] \end{aligned}\\
&\qquad\quad \mathit{expr}_n = \mathit{expr}_f\ \mathit{this}_n\\
&\qquad\quad \mathit{expr}'_n = \mathit{expr}'_f\ \mathit{this}_n
\end{aligned}
$$

Another commonly used type intention is *unknownType* which is defined as follows:

$$
\begin{aligned}
&\mathit{unknownType} :: \mathit{Node}\\
&\mathit{unknownType} \ = \ \mathit{mknode}\ [\ \begin{aligned}[t] &\mathit{isUnknownType} &&= \mathit{True}\\ &\mathit{isEqToType} &&= (\lambda\ \mathit{ti} \rightarrow \mathit{ti}.\mathit{isUnknownType})\\ &\ldots\ ] \end{aligned}
\end{aligned}
$$

Some constructs in PLX can not be safely type checked. For example, the subscript operator on heterogeneous vectors can not always statically calculate its result type since the indexing expression may not be statically known and elements of different types may exist at different indexes in the heterogeneous vector. Thus, type errors are generated when an incorrect type is found; type errors are not generated for correct types or an *unknownType*.

Of course, in order to claim that our typing framework is defined in a modular way, it must be easy to add new types without disturbing the definitions of existing types. To add new types one must simply define a new type intention with an appropriate definition for *isEqToType* and an associated *is...Type* attribute that is defined to true on that intention and defaults to false. Thus, the algorithm type identification and testing of type equality is defined in a distributed manner in the attribute definition and ensures that adding new types does not require the modification of any existing specifications. This provides a high degree of modularity in the definitions of types.

Now that we have seen a couple of example intentions, it is worthwhile to reflect on what has been presented so far. Up to this point, all definitions are merely a method for writing attribute grammars as lazy function programs, an idea that was first put forward by Johnsson [Joh87]. The advantage of this method is that one gets none of the restrictions imposed by special purpose attribute grammar systems. The price of that flexibility is the absence of many static checks, in particular for completeness ("have all attributes been defined?") and circularity ("is there an order in which attributes can be evaluated?"). These deficiencies are not important here, because we do not aim to present an industrial-strength reduction engine. Instead our purpose is to lay bare the essential ideas.

2.2 Forwarding

Forwarding is a technique by which an intention specifies another intention as the place to find values of attributes which it does not explicitly define itself. That is, an intention may specify or create a *forwards–to* node in the AST such that if the intention is queried for the value of an attribute which it does not explicitly define then the query is passed, or *forwarded*, to the designated forwards–to node which will return the attribute value defined for it. The forwards–to node will either explicitly define a value for this attribute or may also forward the query to its own forwards–to node.

Since nodes are records, one way to implement forwarding uses field overriding in records. An intention node n is created from its forwards–to intention node n' by overriding the fields in n' with the corresponding fields explicitly defined by n. For example, if n explicitly defines only the pp attribute and forwards all other attribute value queries to n', then n can be defined as $n \ = \ n' \oplus [\, pp \ = \ \ldots\,]$.

2.2.1 Forwarding for intention definition reuse

An example will help to clarify. Consider adding a **for** loop intention to PLX. We could write a definition similar to that of the while loop intention which explicitly defines all relevant attributes, but since a for loop can be translated into a assignment followed by a while loop it is better to reuse the definition of these intentions. Thus, we will define the for loop such that it forwards to the semantically equivalent while loop construct and only defines the for loop specific attributes explicitly. Its definition is then very simple and compact.

A for loop node has four descendants: one containing the index variable, the starting point of the iteration, the end point, and the body. It is defined as follows:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{for} :: \mathit{SemTree} \to \mathit{SemTree} \to \mathit{SemTree} \to \mathit{SemTree} \to \mathit{Node} \to \mathit{Node}} \\
\multicolumn{4}{l}{\mathit{for}\ \mathit{index}_f\ \mathit{start}_f\ \mathit{end}_f\ \mathit{body}_f\ \mathit{parent}_n \ = \ \mathit{for}_n} \\
\quad \textbf{where}\ \mathit{for}_n & = \mathit{forward}_n \oplus [\, \mathit{pp} & = & \text{“\textit{for} ”} +\!\!+ \mathit{index}_n.\mathit{pp} +\!\!+ \text{“ = ”} +\!\!+ \\
 & & & \mathit{start}_n.\mathit{pp} +\!\!+ \text{“ \textit{to} ”} +\!\!+ \mathit{end}_n.\mathit{pp} +\!\!+ \\
 & & & \text{“ \textit{do} ”} +\!\!+ \mathit{body}_n.\mathit{pp} +\!\!+ \text{“ \textit{end}”} \ ]
\end{array}
$$

$$
\begin{array}{ll}
\mathit{index}_n & = \mathit{index}_f\ \mathit{for}_n \\
\mathit{start}_n & = \mathit{start}_f\ \mathit{for}_n \\
\mathit{end}_n & = \mathit{end}_f\ \mathit{for}_n \\
\mathit{body}_n & = \mathit{body}_f\ \mathit{for}_n \\
\mathit{forward}_n & = \mathit{forward}_f\ \mathit{parent}_n \\
\mathit{forward}_f & = \mathit{parseEnv} \\
 & \qquad [\ (\text{“}\$\mathit{index}\text{”}, \mathit{index}_f),\ (\text{“}\$\mathit{start}\text{”}, \mathit{start}_f) \\
 & \qquad\ \ (\text{“}\$\mathit{end}\text{”},\ \mathit{end}_f),\ (\text{“}\$\mathit{body}\text{”}, \mathit{body}_f\ )\ ] \\
 & \qquad (\text{“}\ \$\mathit{index}\ :=\ \$\mathit{start}\ ; \\
 & \qquad\quad \textbf{while}\ (\ \$\mathit{index}\ \leq\ \$\mathit{end}\ )\ \textbf{do} \\
 & \qquad\quad\ \ \$\mathit{body}\ ;\ \$\mathit{index}\ :=\ \$\mathit{index}\ +\ 1 \\
 & \qquad\quad \textbf{end}\text{”}\ )
\end{array}
$$

Here, a for loop intention defines the value of the pretty print attribute *pp* explicitly, but gets values for all other attributes from the node $\mathit{forward}_n$. For example, when the for loop's parent asks it for the loop's Java byte code attribute *jbc*, the for loop node forwards this query to $\mathit{forward}_n$ and its value for *jbc* is returned to the for loops parent. We define $\mathit{forward}_n$ itself as the root node in the AST of the semantically equivalent assignment and while loop construct.

The function $\mathit{forward}_f$ has type *SemTree*; it is the semantics of the while loop corresponding to the original for loop. The node $\mathit{forward}_n$ is constructed by passing the parent node of the for loop parent_n to $\mathit{forward}_f$. Thus, the parent node of the forwards–to node is the same as the parent of the for intention node.

We do not want to define $\mathit{forward}_f$ by writing out its abstract syntax since it is tedious and difficult to read. Instead we simply write out the construct we wish to forward to as a string and let the parser, which has type $\mathit{String} \to \mathit{SemTree}$, construct the abstract syntax tree. The function *parseEnv* calls the parser to build the AST function for the string given as its last argument. When the parser builds the AST, for each *meta–variable* ($\$\mathit{index}$, $\$\mathit{start}$, $\$\mathit{end}$ and $\$\mathit{body}$) it inserts into the AST the corresponding tree semantics, (index_f, start_f, end_f and body_f). These inserted trees are passed to the parser in the 'environment' provided as the first parameter to *parseEnv*.

2.2.2 An alternative implementation of forwarding

Although the above implementation of forwarding works, it does not follow our intuitive understanding of "forwarding an attribute query to another node." Also, we do not want to write our PLX specification so that we are locked into any single implementation of forwarding. We will instead create a record combinator, $\rhd$, to specify an intention's forwards–to node. We can thus experiment with different implementations of forwarding by changing its definition. As language designers, we want to express our *intention* that one node forwards to another, but we do not want to specify how this is done.

The definition of the for loop intention node for_n given above will instead be written as follows:

$$\mathit{for}_n\ =\ \mathit{mknode}\,[\,\mathit{pp}\ =\ \ldots\,]\ \rhd\ \mathit{forward}_n$$

The implementation of forwarding is as follows.

$$n \rhd fw_n \;=\; n \oplus [\, forwardsTo \;=\; Just\, fw_n \,]$$

It makes use of an attribute *forwardsTo*, which has type *MaybeNode*. Values of type $Maybe\,\alpha$ are either *Nothing* or of the form *Just* v, where v is of type α. The value of the *forwardsTo* attribute is *Nothing* if the node does not forward to another node, or the value is $Just\, fw_n$ if it forwards to the node fw_n. For this to work we must alter the definition of *initNode*, the initial node to which *mknode* adds attribute definitions, as well as the definition of the attribute lookup function (.).

The definition of *initNode* is changed so that its value for *forwardsTo* is *Nothing*. The *forwardsTo* combinator will overwrite this value if it is used in the definition of the intention node, otherwise, it will still have the value *Nothing*.

We also change the definition of the lookup operator (.) to

$$n.a \;=\; \mathbf{if}\, n \text{ `}\mathit{defines}\text{` } a \;\mathbf{then}\, pget\ a\ n \;\mathbf{else}\; n.forwardsTo.a$$

so that it first determines if the node n defines the attribute a; if it does, it uses the primitive field access operation *pget* to retrieve the attribute value, or queries the node for its forwards–to node and then queries that node. The lookup operator (.) is defined to be left associative.

2.2.3 Forwarding and re-targeting compilers

As we have just seen, forwarding makes it very easy to build new intentions from old ones. Now assume that every intention added to PLX forwards to an implementation in terms of more primitive intentions. Then eventually every intention reduces to a small set of core primitives, which one could call *base PLX*. Such a regime makes certain tasks rather easy.

For example, we can more easily re-target our PLX compiler to another target language, say C, by adding to each base PLX intention an attribute definition for its C translation. If all other intentions eventually forward to base PLX intentions, then these intentions will be able to define their C translation without having to change their definition.

We regard such modularity as important since authors of non-base PLX intentions cannot know all the possible future languages to which PLX will be translated. But by ensuring that all intentions eventually forward to base PLX intentions we know that defining a translation for the base language suffices.

2.3 Aspects

In order to achieve a modular language design, it is imperative that it is possible to "package"intentions and attribute definitions in a form convenient to the task at hand. Consider the case in which different parties want to provide attribute definitions for the base PLX intentions to translate PLX into the machine language of a specific micro processor or the byte code of a different virtual machine. But how are such attribute definitions packaged? Certainly one cannot require each party to add their attribute definitions to a base PLX intention by editing the original source code definition of the intention. Each set of attribute definitions for a new translation should be packaged into its own module or component that can be individually imported into a programmer's environment to allow their programs to be translated into the new target language.

We call such components *aspects*.[6] An aspect is a collection of attribute definitions for some set of existing PLX intentions – either base PLX intentions or intentions built on top of them. For example, an aspect defining a translation to C is sketched below.

$$\begin{array}{l}
\ \textbf{aspect}\ \mathit{CTrans}\ : \\
\mathit{while}\ \mathit{cond}_n\ \mathit{body}_n\ \mathit{parent}_n\ \mathit{this}_n \\
\quad = [\ \mathit{cTrans}\ =\ \ldots\ ] \\
\mathit{exprSeq}\ \mathit{expr}_n\ \mathit{expr}'_n\ \mathit{parent}_n\ \mathit{this}_n \\
\quad = [\ \mathit{cTrans}\ =\ \ldots\ ] \\
\ldots
\end{array}$$

The collection of attribute definitions for a single intention in an aspect, such as the C translation attribute for the *while* intention defined in **aspect** *CTrans* **for** *while* above, is a mapping from the nodes in the intentions context, that is, its children, parent and self to a list of attribute definitions of type $\mathit{Node} \rightarrow \mathit{Node}$. These are the same kind of attribute definitions passed to the node constructor *mknode*. Using a simple preprocessor, all aspects for an intention are collected into a global function *aspects*, with type $\mathit{String} \rightarrow [\,\mathit{Node} \rightarrow \mathit{Node}\,]$, which maps intentions names to the list of attribute definitions defined for that intention. Of course, a more sophisticated implementation might allow this collecting to be done *after* each aspect has been compiled.

If each intention definition also defines an attribute *iname* containing its name, we can redefine the node constructor *mknode* as follows to use both the attribute definitions given directly in the intention definition and the attribute definitions given in aspects when constructing a node:

$$\begin{array}{l}
\mathit{mknode}\ \mathit{attrDefs}\ =\ \mathit{node}\ \oplus\ (\mathit{aspects}\ \ \mathit{node.iname}) \\
\ \textbf{where}\ \mathit{node}\ =\ \mathit{initNode}\ \oplus\ \mathit{attrDefs}
\end{array}$$

Here *mknode* first constructs a node consisting of the attribute definitions from the intention definition then gets the intentions *iname* and looks up the aspect attribute definitions and applies them to create the node which it returns.

In previous work we defined a mechanism which provided a Haskell type for aspects and used the Haskell type system to verify that attributes were defined exactly once for each language construct [dMPJVW99]. The definition provided above does not provide such checks; it is a simple syntactic method for collecting and applying aspects.

[6]Many existing attribute grammar systems, such as Eli [GHL+92], provide similar facilities.

2.3.1 Defining attributes in multiple aspects.

One attribute that many intentions and aspects define is *errors*. In our simplified example, it is a list of error messages; its type is $[\,String\,]$, list of strings. The attribute *errors* on an intention node will contain all the errors messages for this node. What happens when more than one aspect for an intention wants to define errors? If each aspect uses the $(=)$ field definition operator then the first definition in the final list of attribute definitions collected from the different aspects will override all other definitions. Instead, we want each attribute definition in the aspects to *contribute* to the final value of *errors*. This will allow each aspect to perform its own error checking and contribute the errors it detects to the set of all errors for an intention.

To do this, we use a different field definition function called *contains* which does not override previous definitions like $(=)$ does, but combines them using an attribute specific function. This function is extracted from the attribute by the *containsOp* function. For example, *containsOp errors* is the list concatenation operator $(++)$. We can thus define *contains*, which has the same type as the attribute definition operator $(=)$, as follows:

$$\begin{aligned}
&contains :: Attr\,\alpha \rightarrow \alpha \rightarrow Node \rightarrow Node\\
&contains\ attr\ value\ node = node \oplus [attr = (combine\ (node.attr)\ value\ )]\\
&\quad \textbf{where}\ combine = containsOp\ attr
\end{aligned}$$

We shall sometimes find it convenient to write

$$attr \supset value$$

in lieu of *contains attr value*. With this operation, we can define several contributing definitions of an attribute in different aspects. For the *errors* attribute, each aspect may do its own error checking and contribute the list of errors it finds the the *errors* attribute. If it finds no errors, it will contribute the empty list ([ ]) to *errors*.

To facilitate error generation, we will use a function *mkErrorUnless* which has the type $mkErrorUnless :: Bool \rightarrow String \rightarrow [\,String\,]$ which returns the empty list of errors if its first parameter is *true* and otherwise creates the singleton list containing its second parameter.

2.4 Further examples of forwarding

We conclude the discussion of PLX and its intentional implementation with a number of further applications of forwarding, as this is the most powerful and unusual feature we have introduced. The first example shows how operator overloading might be modelled using forwarding. Next, it is shown how one can neatly use forwarding to mimic the *syntactic references* found in many attribute grammar systems. Finally, it is shown how default definitions for attributes (such as copy rules) are obtained via forwarding. There is another important application of forwarding, namely the implementation of optimising transformations – that topic will be covered in the next section, where we study the *mkArray* feature.

2.4.1 Forwarding for overloaded operator resolution

An important use of forwarding is to specialise generic or overloaded operators. For example, suppose our language has both numbers and strings and we want to overload the + symbol to mean both addition and string concatenation. We will define a generic + intention which *forwards to* either addition or string concatenation, based on the types of the operands. First *addition* is defined as follows:

$$
\begin{array}{l}
\mathit{addition} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node} \\
\mathit{addition}\ \mathit{expr}_f\ \mathit{expr}'_f\ \mathit{parent}_n\ =\ \mathit{add}_n \\
\quad \text{where } \mathit{add}_n\ = \mathit{mknode}\ [\ \mathit{typei}\ =\ \mathit{numberType}\ , \\
\qquad\qquad\qquad\qquad\qquad \mathit{jbc}\ =\ \ldots\ ] \\
\qquad\quad \mathit{expr}_n = \mathit{expr}_f\ \mathit{add}_n \\
\qquad\quad \mathit{expr}'_n = \mathit{expr}'_f\ \mathit{add}_n
\end{array}
$$

This intention takes arguments expr_f and expr'_f and defines the *typei* and *jbc* attribute, as well as possibly some other attributes. The intention *concat* is defined in a similar fashion, but there *typei* is *stringType* and *jbc* is the Java byte code for string concatenation.

As they stand, a concrete syntax could designate + for addition and ++ for concatenation such that a parser or structure editor would build the AST with these intentions. To overload +, however, we will define a generic intention called *plus* which forwards to either *addition* or *concat* as follows:

$$
\begin{array}{l}
\mathit{plus} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node} \\
\mathit{plus}\ \mathit{expr}_f\ \mathit{expr}'_f\ \mathit{parent}_n\ =\ \mathit{plus}_n \\
\quad \textbf{where}\ \mathit{plus}_n \quad = \mathit{mknode}\ [\ \mathit{errors}\ \supset\ \mathit{mkErrorUnless}\ \mathit{hasPlus} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{“}\mathit{type\ of}\text{ ”} +\!\!\!+\ \mathit{expr}_n.\mathit{pp}\ +\!\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \text{“ }\mathit{doesn't\ overload\ plus}\ (+)\text{”})\ ] \\
\qquad\qquad\qquad\qquad]\ \ \rhd\ \ \mathit{forward}_n \\
\qquad\quad \mathit{expr}_n \quad\ = \mathit{expr}_f\ \mathit{plus}_n \\
\qquad\quad \mathit{expr}'_n \quad\ = \mathit{expr}'_f\ \mathit{plus}_n \\
\qquad\quad \mathit{forward}_n = \mathit{forward}_f\ \mathit{parent}_n \\
\qquad\quad (\mathit{forward}_f,\ \mathit{hasPlus}) = \textbf{case}\ (\mathit{expr}_n.\mathit{typei}.\mathit{plusSpecializer})\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing}\ \rightarrow\ (\mathit{undefined},\ \mathit{False}\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{ps}\ \ \rightarrow\ (\mathit{ps}\ \mathit{expr}_f\ \mathit{expr}'_f,\ \mathit{True})
\end{array}
$$

We query the *typei* intention attribute of the first operand expr_n to determine if its type has an operator for overloading *plus*. The type of the *plusSpecializer* attribute is

$$\mathit{Maybe}\ (\ \mathit{SemTree}\ \rightarrow\ \mathit{SemTree}\ \rightarrow\ \mathit{Node}\ \rightarrow\ \mathit{Node}\ )$$

If the value of *plusSpecializer* is the default, namely *Nothing*, then this type does not overload plus and $\mathit{forward}_f$ is assigned the *undefined* value and we contribute to *errors* a list containing an error message. Otherwise, we set $\mathit{forward}_f$ to the plus specializer function *ps* with the two child expressions specified as its children. The value of *ps* will be either the *addition* or *concat* intentions. This intention definition function is used to create the node which *plus* forwards to. On the *numberType* intention defined above, we will have to add the attribute definition *plusSpecializer* = *Just addition*, and on *stringType*, the attribute definition *plusSpecializer* = *Just concat*. When the *plus* intention is asked, for example, for its Java byte code, it will forward this query to either the *addition* or *concat* intention, depending on the types of the operands and the intended operation, effectively resolving the overloading of the + operator.

By defining operator overloading in this way, any new type intentions added to PLX can specify a *plusSpecializer* to overload the (+) operator without making any change to the definition of the *plus* intention. We will see this when we add the array type in the following section. It is highly undesirable that the addition of new intentions, be they type or language constructs, requires a change to existing intentions in the language. It would be wholly unacceptable if *plus* was defined so that it knew about the different types and operations which overload it. Consider the following non-modular definition of $forward_f$

$$\begin{aligned} forward_f \;=\; & \textbf{if}\; expr_n.typei.isNumberType\; \textbf{then}\; addition\; expr_f\; expr'_f \\ & \textbf{else if}\; expr_n.typei.isStringType\; \textbf{then}\; concat\; expr_f\; expr'_f \\ & \textbf{else}\; undefined \end{aligned}$$

If $forward_f$ were defined in this way, then adding the array addition intention so that it overloads the (+) operator would require a change to the *plus* intention and violate the modular composition of languages which is the goal of intentional programming.

It is worthwhile to note that unlike our earlier examples of forwarding, the choice of the forwards-to node is made at runtime. Mernik has suggested the use of inheritance to compose attribute grammar components, and many applications of forwarding are covered by his technique. Examples such as the one above where the choice is made dynamically are however somewhat difficult to model using static inheritance.

2.4.2 Forwarding to link references to declarations

A similar technique can be used to specialise a variable reference to its type and to include its declaration in its context. We have chosen to depart significantly from the R5.0 approach to this issue here; we discuss this further at the end of this paper. A reference to a variable of type *numberType* can be defined as follows:

$$\begin{aligned} & refToNumber :: Node \to Terminal \to Node \to Node \\ & refToNumber\; dcl_n\; id_n\; parent_n \;=\; ref_n \\ & \quad \textbf{where}\; ref_n \;=\; mknode\; [\; name \;=\; id_n.lexeme, \\ & \qquad\qquad\qquad\qquad\quad typei \;=\; dcl_n.typei, \\ & \qquad\qquad\qquad\qquad\quad jbc \;=\; \ldots\; dcl_n\; \ldots\;] \end{aligned}$$

Here the reference's declaration is assumed to be in its context since it appears as the first parameter in its intention definition function. In this case, we query the declaration dcl_n for its type and use that as the type of the variable reference. (In this case, we could simply define *typei* to be *numberType*, but in the case of compound types such as *array*, we wouldn't know the component type and in those cases we must get the type from the declaration.) The dcl_n can also be used in defining the Java byte code as it provides, for example, the stack offset this variable will have in the runtime stack. In general, this provides a very convenient mechanism to access attributes on a variable's definition and is much simpler than packaging all the relevant information into an *environment* and passing this information to the the variable reference by passing it up to the enclosing *let* and then down the *let* body to the variable reference.

Reference intentions specific to a particular type are not put into the AST directly by a parser or structure editor directly, but, like the specialised *addition* and *concat* intentions, appear in the AST as intentions forwarded to by a generic variable reference which is defined as follows:

$$\begin{array}{l}
genRef :: Terminal \rightarrow Node \rightarrow Node \\
genRef\ id_n\ parent_n\ =\ ref_n \\
\quad \textbf{where}\ \begin{array}[t]{ll}
ref_n & = forward_f\ parent_n \\
dcl_n & = lookup\ (parent_n.env)\ (id_n.lexeme) \\
refSpec_f & = dcl_n.typei.refSpecializer \\
forward_f & = refSpec_f\ dcl_n\ id_f
\end{array}
\end{array}$$

The generic reference uses the lexeme of the identifier $id_n.lexeme$ and the function *lookup* to find the declaration of this reference in the environment *env* of the parent node $parent_n$. The environment attribute is an association list of identifier names and their declaration nodes; it has type $[\,(String,\ Node\,)\,]$. This list is ordered so that local declarations appear before global declarations in the list and thus a front to back search for a declaration enforces the standard nesting scope rules of PLX and returns the correct variable declaration. We query the declaration for its type which we then query for its reference specializer attribute. This attribute, *refSpecializer*, is an intention building function of type $Node \rightarrow Terminal \rightarrow Node \rightarrow Node$. Using *refSpecializer*, we create the node that *genRef* forwards to by giving it as parameters the node dcl_n, the *Terminal* id_n and the parent node $parent_n$.

Declaration intentions are also specialised by their type and are treated in a similar fashion except that there are two generic declaration intentions, *genLetDcl* for let bindings and *genLambdaDcl* for lambda bindings. These forward to type and binding specific declaration intentions and thus each type provides a lambda declaration intention and a let binding declaration intention. For example, the *function* type will define the intentions *letDclForFunction* and *lambdaDclForFunction*. *letDclForFunction* is similar to specialised references like *refToNumber* in that is has an extra node in its context; in this case it is the defining expression. It is defined as follows:

$$\begin{array}{l}
letDclForFunction :: Node \rightarrow Terminal \rightarrow Node \rightarrow Node \\
letDclForFunction\ expr_n\ id_n\ parent_n\ =\ dcl_n \\
\quad \textbf{where}\ dcl_n\ = mknode\ [\ \begin{array}[t]{ll}
name & = id_n.lexeme, \\
typei & = expr_n.typei, \\
jbc & = \ldots\ expr_n\ \ldots\ ]
\end{array}
\end{array}$$

The *genLetDcl* has access to the defining expression which it queries for its type intention in order to define its *typei* attribute. Besides *refSpecializer*, a type intention must define the *letDclSpecializer* attribute of type $Node \rightarrow Terminal \rightarrow Node \rightarrow Node$ and the *lambdaDclSpecializer* attribute of type $Terminal \rightarrow Node \rightarrow Node$. The values of these attributes are the binding specific intentions of that type; for the *function* type intention their respective values are *letDclForFunction* and *lambdaDclForFunction*. As with the generic reference *genRef*, *genLetDcl* uses the value of the attribute *letDclSpecializer* to build the node it forwards to by giving the intention building function the defining expression node, the *Terminal* id_n and node $parent_n$. The *lambdaDclForFunction* intention is defined in a similar way except that there is no defining expression in its context. A standard type inference mechanism is used to determine the types of the lambda bound free variables so that the generic lambda declaration knows which type to query for the specializer functions.

2.4.3 Forwarding to provide default definitions

It is often helpful to be able to provide default definitions for attributes. Forwarding provides a convenient mechanism for defining default definitions for attributes. A question immediately arises however: if an intention does not define and attribute which has a default value but forwards to an intention which defines this attribute, which value should be used? In terms of forwarding, this question can be restated as follows: are default definitions placed at the beginning or end of the "forwards to chain" of an intention?

Neither option is always the correct one, so we allow the intention author to answer this question as they see fit for each particular attribute. We will refer to default definitions which take precedence over definitions on forwarded to nodes as *immediate* default definitions. Those which are used only if the (direct or indirect) forwards to nodes do not define the attribute referred to as *final* default definitions. Immediate default definitions are placed at the beginning of a node's forwards to chain and final default definitions are placed at the end.

We thus define below two intentions, *immedDefs* and *finalDefs*, which other intentions will forward to in order to have access to default definitions of attributes.

$$
\begin{array}{l}
immedDefs :: Node \rightarrow [\,Node\,] \rightarrow Node \rightarrow Node \\
immedDefs\ self\ children\ parent_n \ = \ immdefs_n \\
\quad \textbf{where}\ immdefs_n \ = \ mknode\ [\ name = \text{“}immedDefs\ for\text{ “} +\!\!+\ self.name\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad env \ = \ parent_n.env\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad isNumberType \ = \ False, \\
\qquad\qquad\qquad\qquad\qquad\qquad plusSpecializer \ = \ Nothing\ ]
\end{array}
$$

$$
\begin{array}{l}
finalDefs :: Node \rightarrow [\,Node\,] \rightarrow Node \rightarrow Node \\
finalDefs\ self\ children\ parent_n \ = \ findefs_n \\
\quad \textbf{where}\ findefs_n \ = \ mknode\ [\ name = \text{“}finalDefs\ for\text{ “} +\!\!+\ self.name\ , \\
\qquad\qquad\qquad\qquad\qquad\qquad pure \ = \ and\ (map\ (get\ pure)\ children)\ ]
\end{array}
$$

Notice that these intentions have a slightly different type from the ordinary intentions we have studied so far: the first argument is the intention node we are providing default answers for and the second is the list of child nodes, not a list of type [*SemTree*]. This type is reminiscent of the types of aspects. These intentions takes the forwarding intention, the list of its children, and its parent as parameters and defines attribute values based on other attribute values on these parameter nodes. If the forwarding intention doesn't define an attribute value, the query for that value is forwarded to the a default intention node. In this case, the immediate default intention defines the default environment *env* of an intention to be the environment of its parent. Since most intentions do not access or change the environment, if they forward to the immediate default intention then they can leave out any reference to *env* and its value will still be passed down the abstract syntax tree as it should be. This is the same as the mechanism for default copy of inherited attributes to children in many attribute grammar systems.

We can also define default rules for attributes values that move up the tree. The attribute *pure* is true on an intention if its evaluation does not produce any side effects. The assignment intention must explicitly define this attribute to be *false* but others can rely on the default definition which maps the function *get* over the list of child nodes to create a list of Boolean values and then takes their conjunction. The function *get*, is defined as $get\ attr\ node\ =\ node\ .\ attr$ and has type $get\ ::\ Attr\ \alpha\ \rightarrow\ Node\ \rightarrow\ \alpha$. Note that we have defined the final default for *pure*, and not its immediate default. We want to use this default definition only if none of the forwarded to intentions define it. Consider the **for** intention defined above. If we defined an immediate default for *pure* similar to the one above, a for loop with no assignment statements would have a *pure* value of false. But, since it forwards to a construct which contains an assignment statement this would be erroneous.

Since type intentions will also forward to an instance of these default intentions, the default definition for the attribute *isNumberType* is given here as well. Each type intention must define a similar attribute and provide a default definition which evaluates to *false*. Thus, type intentions which forward to the immediate default intention are not required to have any knowledge about the existence of any other types to be able to distinguish themselves from them and will answer *false* when asked if it is any type other than its type. This provides a high degree of modularity in the definitions of types. Similarly, the intention *plus* defines the immediate default of *plusSpecializer* to be *Nothing*. Its use was discussed in Section 2.4.1.

To make use of the defaulting mechanism, we have to modify the intention definitions we have shown so far so that they forward to the immediate and final default intentions. We modify the for intention to read as follows:

$$\begin{array}{l}
for\ index_f\ start_f\ end_f\ body_f\ parent_n\ =\ for_n \\
\quad \textbf{where}\ for_n\ =\ mknode\ [\ldots \\
\qquad\qquad\qquad\qquad]\ \rhd\ immedDefs\ for_n\ [index_n,\ start_n,\ end_n,\ body_n]\ parent_n \\
\qquad\qquad\qquad\qquad\quad \rhd\ forward_n \\
\qquad\quad forward_n\ =\ \ldots
\end{array}$$

We see that attribute queries to the for intention are forwarded first to the immediate defaults intention and then to the assignment and while loop construct in the $forward_n$ intention. The operator $\rhd$ is right associative. To the intentions above which do not already explicitly forward to another intention we should add the clause

$\ldots \; \triangleright \; \mathit{immedDefs}\ \mathit{this}_n\ [\ldots\ \mathit{list\ of\ child\ nodes}\ \ldots]\ \mathit{parent}_n$
$\qquad \triangleright \; \mathit{finalDefs}\ \mathit{this}_n\ [\ldots\ \mathit{list\ of\ child\ nodes}\ \ldots]\ \mathit{parent}_n$

after the list of attribute definitions passed to *mknode*.

Remarks: Since *immedDefs* and *finalDefs* are intentions, we can, and should, provide attribute definitions for them by using aspects instead of writing attribute definitions in the intention definition as we have done above. The immediate default for *env* could thus be more appropriately defined as

aspect *Environment* :
$\mathit{immedDefs}\ \mathit{self}\ \mathit{children}\ \mathit{parent}_n\ \mathit{immedefs}_n$
$\quad = [\ \mathit{env}\ =\ \mathit{parent}_n.\mathit{env}\ ]$

Also, the forwarding clauses above admittedly seem rather verbose. Default values and definitions are typically something one gets without writing down anything, yet above we have written a fair amount. Our aim here is to make clear how the defaulting mechanism works and we do this by being more explicit than we need to be. In fact, we can make the application of immediate default definitions automatic by making a few small changes to *mknode*. If all the immediate default definitions are made in aspects, then these definitions can be collected in a manner similar to the aspects of other intentions to create a list of $\mathit{Node} \rightarrow \mathit{Node}$ default attribute definitions that can be applied directly by *mknode*.

As witnessed by the possible errors introduced to the for intention by providing an immediate default definition for the *pure* attribute, the use of default definitions is not always straight forward. In fact, it has been our experience that if default definitions can be avoided, it is best to do so. For the *pure* attribute, it is better to explicitly define *pure* on all of the base PLX intentions and have no default *pure* definitions.

2.5 Circular definitions

The solutions to several problems in language processing are often concisely and naturally stated by a set of recursive (circular) definitions. Knuth however did not allow circular attribute definitions in his original design of attribute grammars, and restating these solutions in their equivalent non-circular form requires the use of additional attributes and more complex attribute definition functions. Fortunately Farrow [Far86] showed that under some conditions circular attribute definitions are well-founded and that the least fix-point computations they represent are computable by successive approximation. Circular definitions are incorporated into our model of reduction and their inclusion was inspired by the notion of rollback in R5.0. In this section we show how the attribute *pure*, which is *true* on constructs which have no side effects, is defined using circular attribute definitions.

Consider the following program:

let $c\ =\ 0,$
$\quad f\ =\ \lambda\, i\ \rightarrow$ **begin** $c\ =\ c\ +\ 1;$

```
                 if $i = 0$ then 0 else $i + f(i - 1)$ end
in  $mkArray(10, \lambda i \rightarrow f(i) + f(10))$
end
```

In general, a construct is pure if it isn't an assignment statement and its children are pure. Most intentions use the default definition of *pure* we saw above which encodes this fact. The variable reference intentions are interesting examples of intentions which don't follow this trend. The generic reference intention does not define *pure* but forwards any *pure* queries to the variable reference intention specialised by its type, *e.g.* the *refToNumber* and *refToFunction* intentions. Clearly, the *refToNumber* should define *pure* to be *true*, as we see in the aspect *pure* below. But what about references to functions? Here we find another use of the reference's link back to its declaration and use it to define that a *refToFunction* is pure if its declaration dcl_n is pure. A lambda binding is safely assumed to be not pure since there is no defining function body to examine. A *letDclForFunction* is defined to be pure if its defining function, $expr_n$, is pure and it is not assigned to. In PLX it is possible to assign a new functional value to a function variable; if this happens we can not statically know that the function variable contains a pure function. The attribute *assignedToDcls* contains the list of (type-specific) declarations which may be assigned to in the program; its definition as an aspect is discussed in Section 3.2.3 This test is overly conservative; a better solution is mentioned in Section 5.3 Below is part of the *pure* aspect:

```
aspect $pure$ :
$assign\ var_n\ expr_n\ parent_n\ this_n$
  $= [\ pure\ =\ False\ ]$
$refToNumber\ dcl_n\ id_n\ parent_n\ this_n$
  $= [\ pure\ =\ True\ ]$
$refToFunction\ dcl_n\ id_n\ parent_n\ this_n$
  $= [\ pure\ =\ dcl_n.pure\ ]$
$lambdaDclForFunction\ id_n\ parent_n\ this_n$
  $= [\ pure\ =\ False\ ]$
$letDclForFunction\ expr_n\ id_n\ parent_n\ this_n$
  $= [\ pure\ =\ expr_n.pure\ \wedge$
                 $this_n \notin this_n.assignedToDcls\ ]$
```

In the case of recursive functions declared in let bindings, as in the example above, we get a circular set of *pure* attribute definitions; the function is pure if its body is pure, but its body, which contains a reference to the function, is pure if the declaration is pure. Although circular, the resulting set of defining equations does have a unique solution and we can compute it by calculating successive approximations. This is done by assigning each attribute *attr* instance in the circularity with an initial value called "bottom" and written as $\perp_{attr}$. In the case of *pure*, $\perp_{pure} = true$. We then compute new values for each attribute instance in the circularity based on the current attribute values. We repeat this process until, for each attribute value, the new value is the same as the previous value.

In the example above, the set of circular *pure* attribute definitions contains all of those definitions for *pure* in the declaration and body of f except the assignment statement and its children.The references to f in the *mkArray* are also not part of the circularity. To these circular instances we assign the initial value of *true*. We then compute, in an unspecified order, new values for these attributes based on the previous ones. We calculate the first subsequent value of *pure* on the function reference f to be the current value of *pure* on the declaration, that is, *true*. The calculation of subsequent new values of *pure* for all other attributes in the circularity results in *true* as well, except for the *exprSeq* intention. Its subsequent new value is *false* since it computes this value as the conjunction of the *pure* values of it children and its first child is the assignment statement which defines *pure* to be *false* and is not part of the circularity. Since the value on *exprSeq* changed from *true*, the initial value, to *false*, we calculate new values for all these *pure* attributes in the circularity. After enough calculations, the value of *pure* on the circular attributes, the function declaration, and the function reference all get the value of *false*. When the new values of *pure* are the same as the old values, this successive approximation process stops.

More efficient evaluation strategies for circular definitions can be found in papers by Farrow [Far86] and Long [Jon90] and such strategies would be employed by an industrial-strength implementation of reduction. Farrow [Far86] also provides a set of conditions which guarantee the termination of the iterative computation described above. First, the domain of attribute values must be a complete partial order in which it is possible to test for equality. Secondly, the attribute functions must be monotonic and satisfy the ascending chain condition. The first condition ensures that the check for termination of the successive approximation loop is possible and that there is an ordering on possible attribute values. The second condition ensures that the sequence of attribute values which result from the repeated application of the attribute definition functions increases with respect to this ordering to a point where the values stop changing. That is, the successive approximation loop terminates. For more details we refer the reader to Farrow [Far86] and Chirica and Martin [CM97] since our goals here are just to lay out the capabilities of reduction.

These circular definitions do not embed themselves into Haskell as easily as the rest of the definitions in our proto-type. Although Haskell does in effect implement a least fixpoint solution to the attribute definition equations, it is not the correct one. The partial order $\leq_H$ used by Haskell only relates degrees of "definedness" with a bottom value of *undefined*. The partial order we need for circular attributes must relate defined values. Our prototype implements circular attribute definitions via a preprocessor which replaces a single circular attribute definitions with a collection of supporting definitions which explicitly perform the successive approximation algorithm sketched above.

2.6 Inherited attributes

In attribute grammars, *inherited attributes* are those which a node does not define itself but are defined by the node's parent for it. This provides a mechanism for a different value to be assigned by the parent to different children. In R5.0, child nodes ask their parent a question and the parent can examine a "who is asking" parameter to determine which child is asking the question and thus respond differently to different children. How are similar capabilities handled in our model? We provide a slightly different solution in our proto-type. A parent node can provide different "views" of itself to different children by overriding certain attributes on itself with different values and using these overridden versions as the parent of the children. Consider the example of a pretty printing attribute which uses an attribute *ppIndent* which tells it how many spaces its pretty printed text is to be indented.[7] On the *while* intention we will indent the pretty printed text of the condition $cond_n$ by 6 characters (5 for the "while" and one for a following space) more than the *while* and we will indent the pretty printed text of the body $body_n$ by 3 characters more than the *while*. When the $cond_n$ and $body_n$ ask their parent for their *ppIndent* attributes, they will receive different values. The implementation of this in our model is shown below for the *while* intention.[8]

$$
\begin{aligned}
&\mathit{while}\ \mathit{cond}_f\ \mathit{body}_f\ \mathit{parent}_n\ =\ \mathit{this}_n\\
&\quad \textbf{where}\ \mathit{this}_n\ =\ \mathit{mkNode}\,[\,\mathit{ppIndent}\ =\ \mathit{parent}_n.\mathit{ppIndent}\ ,\\
&\qquad\qquad\qquad\qquad\qquad\quad \ldots]\ \rhd\ \ldots\\
&\qquad\qquad \mathit{cond}_n =\ \mathit{cond}_f\ (\mathit{this}_n\ \oplus\ [\,\mathit{ppIndent}\ =\ \mathit{this}_n.\mathit{ppIndent}\ +\ 6\,])\\
&\qquad\qquad \mathit{body}_n =\ \mathit{body}_f\ (\mathit{this}_n\ \oplus\ [\,\mathit{ppIndent}\ =\ \mathit{this}_n.\mathit{ppIndent}\ +\ 3\,])
\end{aligned}
$$

The operator $\oplus$ is the field overriding operator we saw in Section 2.1.2. There is a discrepancy between how this implementation and R5.0 handle forwards-to intentions that define additional "inherited" attributes. The additional inherited attributes are not visible on the trees under the forwarding node as they are in R5.0 but they are available on the trees under the forwards-to node. A mechanism does exist in our model to remove this discrepancy and involves getting the overriding attribute definitions from the forwarding node and applying them to the trees under the forwarding node. We are, however, still investigating the intricacies of these different approaches.

In attribute grammars, a collection of inherited attributes is passed to the child node instead of passing the parent node with all its attributes as is done above. The collection of inherited attributes passed to the child is simply a finite mapping from attribute names to attribute values. But this is in fact exactly how we describe the type *Node* in the beginning of Section 2.1. In both cases a collection of attribute values of type *Node*, called either "inherited attributes" in attribute grammars or "$parent_n$" in our model, is passed to the child. Also, in most attribute grammar systems, the inherited attribute values are automatically assigned as the attribute values of the child node. Thus, in our example instead of asking the parent for *ppIndent*, the value for this attribute is referenced by simply stating $this_n.ppIndent$ instead of $parent_n.ppIndent$. The attribute definition of $ppIndent = parent_n.ppIndent$ seen in *mkNode* above is not needed. Our model also supports the definition of attributes in the style of inherited attributes from attribute grammars which make the assignment of inherited attributes to the child nodes automatically. A change to *mkNode* makes this possible.

[7]The pretty printing attribute *pp* we have seen before does not use this type of information. In fact it is more like a simple "un-parser" than a pretty printer.

[8]A slightly more complicated technique is used to do the same thing on aspects but for brevity we do not present it here.

3 Extending PLX with array expressions

Now that we have illustrated the main concepts of intentional programming with the base language PLX, we can consider how a new intention can be added in a modular manner. In particular, we shall study the introduction of a new type of arrays, along with array operations and optimisations, of the kind that were described in the introduction.

As we saw in the preceding sections, arrays are implemented as vectors and array operations are implemented in terms of vector operations. This is achieved via forwarding: array intentions forward to operations over vectors. There are three reasons why we wish to add arrays to PLX even though it already has vectors:

1. *to raise the level of abstraction:* The *mkArray* syntax is more concise than the syntax of the corresponding vector operations. We are essentially codifying a programming idiom of allocating and initialising vectors.

2. *to increase safety:* The array intentions perform the necessary type checking before forwarding queries for "implementation" attributes to the vector operations. They also implement run–time array bounds checks.

3. *to increase efficiency:* As we will see, the array intention lifts expressions which have an invariant value for each array index from the array initialisation function to an enclosing *let*-binding to avoid needlessly re–evaluating the lifted expression.

The latter two points are of course closely related to raising the level of abstraction. They are also the reason that a simple macro-processor cannot give the same benefits. Safety could be more directly achieved in a language that already has a sufficiently rich type system to express the typing constraints we seek, but the goal is to show how additional error checking capabilities, in this case better type checking, can be incorporated into a set of intentions. Although the type safety constraints we describe are easily expressible in modern strongly typed languages, it is impossible to also specify the associated optimisations even in today's most advanced programming languages.

The structure of this section is as follows. First in Section 3.1 we introduce the basic array intentions, including the safety checks. Next in Section 3.2 we consider how optimisations can be made by certain hoisting transformations.

3.1 The array intentions

Adding arrays to PLX involves not merely one intention, but a whole package of them: we need a notion of array types, the *mkArray* operator for creating new arrays, as well as operations for subscripting, size, pointwise addition, and so on. One could say that such a package of intentions defines a domain-specific language that is added to the host language PLX. Ideally, there are not too many dependencies on the features of PLX, and the package of arrays should not interfere with other possible extensions of PLX.

3.1.1 The array type intention

We begin by introducing the type intention for arrays as follows:

```
arrayType :: Node → Node
arrayType $paramType_n$ = $this_n$
  where $this_n$ = mknode [
                 isArrayType = True,
                 isEqToType
                   = (λ ti → if ti.isArrayType
                               then (ti.compType.isEqToType) $paramType_n$
                               else False ) ,
                 compType = $paramType_n$,
                 refSpecializer = arrayRef,
                 letDclSpecializer = letDclForArray,
                 lambdaDclSpecializer = lambdaDclForArray,
                 plusSpecializer = Just arrayPlus,
                 subscriptSpecializer = Just arraySub
                   ] ▷ (defaults $this_n$ [$paramType_n$] initNode)
```

This type constructor takes as its parameter $paramType_n$ the type of the array elements. Like all type intentions, it defines its own type identification attribute *isArrayType* which is defined to be true on array type intentions and, via the default attribute definition *isArrayType* = *False*, is false on all other intentions. It also defines the type comparison function attribute *isEqToType* which specifies that two arrays have the same type if they are both arrays and their component types *compType* are equal. The type of the array's elements is stored in the *compType* attribute. Unlike heterogeneous vectors whose elements may be of different types, arrays are homogeneous and all elements must be of the same type. We also define the reference and declarations specializer attributes we saw in Section 2.4.2 and the plus specializer from Section 2.4.1 so that we can overload the + operator with array point-wise addition. We also define the *subscriptSpecializer* attribute to be *Just arraySub* so that the array subscript operator can overload the generic subscript operator.

3.1.2 The array creation intention

The *mkArray* intention is used to allocate and initialise an anonymous array in an expression. It takes a size expression *size* and an initialising function *init* defined by a lambda expression. The domain of *init* is the index values from 0 to $size - 1$ and is used to calculate initial values for the array. For example, *mkArray* (10, $\lambda\, i \rightarrow i * i$) creates an array of size 10 with indices 0 ... 9 with values 0, 1, 4, 9, ..., 81.

The implementation of the *mkArray* intention shows how arrays are realised by means of vectors:

```
mkArray :: SemTree → SemTree → Node → Node
mkArray $size_f$ $init_f$ $parent_n$ = $array_n$
  where $array_n$ = mknode [
                     typei = arrayType ($init_n$.typei.returnType) ,
                     pp = "mkArray(" ++ $size_n$.pp ++ "," ++ $init_n$.pp ++ ")"
```

```
              ] ▷ (vec_f parent_n)
size_n   = size_f array_n
init_n   = init_f array_n
vec_f    = parseEnv [("$size", size_f), ("$init", init_f)]
              ( " let  s := $size , i := 0,
                     w := vec(2); v := vec(s)
                in while i < s do
                     v [i] := $init ( i ) ;
                     i :=  i + 1 ;
                   end ;
                   w [0] := s ; w [1] := v ;
                   w
                end" )
```

This intention has two children, namely the expression $size_n$ that defines the array's size, and the initialisation function $init_n$. The type of a *mkArray* intention is *arrayType* with the component type being the type returned from the initialisation function $init_n$. The type of $init_n$ if *functionType* which defines the attribute *returnType*. The *mkArray* intention also defines its own pretty printing attribute *pp*. Queries for all other attributes are forwarded to a *let*-binding that implements the array construction. This *let*-binding is slightly different from the example given in the introduction. We represent an array as a vector of size 2, named w above, which holds the size of the array in position 0 and in position 1 is a reference to another vector, v, which is the same size as the array and contains the values of the array. By storing the size of the array in its vector representation, we can implement run–time array bounds checking and overload the (+) operator.

At this point we should make a few comments about primitive and compound types in PLX. The primitive types *numberType* and *booleanType* occupy one word of storage and use a *pass by value* semantics for parameter passing and assignment. Compound types are built on top of vectors and use a *pass by reference* semantics. Thus, we do not make any reference to the size of the component type of the array in the translation of *mkArray* to vector operators. The vector elements of the vector v will be single word values regardless of the component type of the array. If the component type is a primitive type then only one storage word is used. If it is a compound type only one word is used for the reference to the compound values.

The forwarding function vec_f, of type *SemTree*, is given the same parent as the array intention to create the node that the *mkArray* forwards to. The local variables in the *let*-binding (s, i, v and w) are merely place holders: the parser will generate fresh identifiers when parsing this piece of code. Here and in other places, we are omitting a number of static checks to keep the specification to a reasonable volume. For example, one might want to verify that the initialiser of *mkArray* is truly of function type. Adding the check to the intention above is very straight forward.

3.1.3 The array subscript intention

Since arrays are implemented as vectors we will implement the array subscript operator using vector operators. For example, consider an array a that is represented by a vector $w = a_v$ in the manner defined above by the *mkArray* intention. Then the expression $a[3]_a$, where $[\_]_a$ denotes the array subscript operator, is transformed into the expression $a_v[1]_v[3]_v$, where $[\_]_v$ is the vector subscript operator. In position 1 of the vector a_v is the reference to the vector containing the data of array a. We index this vector at position 3 to retrieve the value of array a at position 3.

We define a generic subscript operator in an identical fashion to the generic *plus* operator so that vector subscript ($[\_]_v$) and array subscript ($[\_]_a$) can both overload this operator in the same way that *addition* and *concat* overload (+). The generic subscript intention *subscript*, written as ($[\_]$), has the same type as *plus* and the vector and array subscript intentions; this type is

$$SemTree \rightarrow SemTree \rightarrow Node \rightarrow Node$$

Like *plus*, *subscript* queries the type of its first child, a vector or array reference, for its *subscriptSpecializer* attribute (instead of the *plusSpecializer* attribute) to build its forwards-to node. The type of the *subSpecializer* attribute is type

$$Maybe\ (\ SemTree \rightarrow SemTree \rightarrow Node \rightarrow Node)$$

with a default value *Nothing*. As we have already seen, *arrayType* carries the attribute definition *subscriptSpecializer* = *arraySub*. Similarly, the vector type is augmented with the definition *subscriptSpecializer* = *vectorSub*.

The array subscript intention itself is defined as follows:

```
arraySub :: SemTree → SemTree → Node → Node
arraySub array_f expr_f parent_n = this_n
  where this_n = mknode [
                    pp = array_n.pp ++ "[" ++ expr_n.pp ++ "]_a",
                    typei = if array_n.typei.isArrayType
                            then array_n.typei.compType
                            else undefined,
                    errors ⊃ mkErrorUnless
                               (array_n.typei.isArrayType ∨
                                array_n.typei.isUnknownType )
                               ( array_n.pp ++ " must have type ArrayType"
                                 ++ " or UnknownType." )
                             ++
                             mkErrorUnless
                               (expr_n.typei.isNumberType ∨
                                expr_n.typei.isUnknownType )
                               (expr_n.pp ++ " must have type "
                                 ++ "NumberType or UnknownType." )
```

```
            ] $\rhd_f$ $vec_n$
$array_n$ = $array_f$ $this_n$
$expr_n$ = $expr_f$ $this_n$
$vec_n$  = $parseEnv$ [(“\$array”, $array_f$), (“\$expr”, $expr_f$)]
            “($toVector$ (\$array)) [ 1 ]$_v$ [ \$expr ]$_v$”
```

Since arrays are homogeneous, the type of an array subscript expression is the component type of the array. This differs from vector subscripts which have type *unknownType* since vectors are heterogeneous and we thus can not always know the type of its components at compile time. This intention also reports an error if the $array_n$ intention is not of type *arrayType* or *unknownType* or if the indexing expression is not of type *numberType* or *unknownType*. One of our goals of adding arrays was to increase the strength of the type checking. We appear to compromise this goal by not generating errors when and child of *arraySub* has type *unknownType*, but we could easily add a *warnings* aspect which generates warning messages when *unknownType* intentions are used. Thus, the absence of such warnings would indicate a program which is guaranteed to be type correct.

The array subscript intention forwards to the vector subscript operations discussed above. Some readers may, however, have expected to see the string

“ $\$array[1]_v[\$expr]_v$ ”

instead of the one above containing the *toVector* expression. The expression $array_n$ has type *arrayType* and the vector subscript operator $[\_]_v$ expects its first operand to have type *vectorType*. So the expression without *toVector* would not be type correct. But the key point is that we expect that any attribute query which the vector subscript operation makes to the array expression *array* will be forwarded to its vector implementation, which does have type *vectorType*, and thus there is not an actual type error.

We introduce the *toVector* intention which is solely used in the construction of forwards–to nodes for array intentions. It is used as a “wrapper” for array intentions to define the *typei* of the array expression to be a *vectorType*.

```
toVector :: SemTree → Node → Node
toVector $expr_f$ $parent_n$ = $this_n$
  where $this_n$ = mknode [
                 typei = vectorType unknownType
               ] $\rhd$ $expr_n$
        $expr_n$ = $expr_f$ $parent_n$
```

This intention forwards all attribute queries, except for the attribute *typei*, to the array expression $expr_n$. This array intention will forward all attribute queries that the vector subscript intention will make to its vector equivalent. Any attributes which the array intention defines explicitly are not of interest to the vector subscript operation and will presumably not be asked for. Note that even though $expr_f$ is a parameter to *toVector* it is not used to create a child *Node* to this intention but to create its forwards-to node.

Summarising the above technicalities, *toVector* simply plays to role of a “wrapper” intention that defines the value of one attribute value and retrieves all other attribute values from the construct it forwards to. Leaving this intention out of the above definition does not change the results of any compilation, it only changes the errors which are reported by ensuring that the spurious error generated by the vector subscript in this instance is removed.

3.1.4 The array size intention

It is useful to have a *size* operator that returns the number of elements in an array. This makes it possible, for instance, to write generic routines for bitmap manipulation, independent of the size of the bitmap. Given the above examples, this intention is straightforward to define, and readers who wish to exercise their understanding of the material so far may wish to construct a definition for themselves before reading on.

The size intention has only one parameter, which is an expression of array type:

```
$arraySize :: SemTree \to Node \to Node$
$arraySize\ expr_f\ parent_n = this_n$
  where $this_n = mknode\ [typei = iNumberType,$
                          $pp = \text{“}size\ (\text{”} +\!\!+ expr_n.pp +\!\!+ \text{“})\text{”}$
                          $] \triangleright vec_n$
        $expr_n = expr_f\ this_n$
        $vec_n = parseEnv\ [(\text{“}\$expr\text{”}, expr_f)]$
                 $\text{“}\ (toVector(\$expr))\ [\,0\,]_v\ \text{”}$
```

For brevity, we have omitted the check that the argument is indeed of array type.

3.1.5 The array subscript intention, revisited

The above definition of array subscript does not perform any run–time bounds checks on the array access. Part of the reason we represent arrays as vectors which store the size of the array is to perform such checks. Now that we have seen how the array size intention works, we can define an alternative array subscript intention which performs these checks. This new intention is defined as follows:

```
$arraySubRtc :: SemTree \to SemTree \to Node \to Node$
$arraySubRtc\ array_f\ index_f\ parent_n = this_n$
  where $this_n = mknode\ [$
                    $pp = array_n.pp +\!\!+ \text{“}\,[\,\text{”} +\!\!+ index_n.pp +\!\!+ \text{“}\,]_{artc}\,\text{”}$ ,
                  $] \triangleright vec_n$
        $array_n = array_f\ this_n$
        $index_n = index_f\ this_n$
        $vec_n = vec_f\ parent_n$
        $vec_f = parseEnv\ [(\text{“}\$a\text{”}, array_f), (\text{“}\$i\text{”}, index_f)]$
                 ( “ if $size\ (\$a) > \$i$ then $\$a\,[\,\$i\,]_a$
                     else $halt$ “array index out of bounds” end ” )
```

This array subscript intention performs the run time bounds check by forwarding to the original array subscript operation wrapped in an if–then–else which performs the check, accesses the array if the index is in range and halts with an error message if it fails the bounds test.

3.1.6 The array addition intention

In the definition of the array type intention *arrayType* we specified that the *plus* specializer attribute *plusSpecializer* on this type is *arrayAdd* in order to overload the (+) operator with array addition. Recall that the overloaded *plus* intention will query the type of its first operand for the intention definition function it is to use to construct the intention which it forwards to. If that operand has type *arrayType*, then *arrayAdd* is the intention which *plus* will forward to. This provides a good example of the modularity we seek in our language design. An intention author can add these array intention definitions to the existing language framework to take advantage of operator overloading without making any changes to the existing language intentions.

We can define *arrayAdd* as follows:

```
$arrayAdd :: SemTree \rightarrow SemTree \rightarrow Node \rightarrow Node$
$arrayAdd\ expr_f\ expr'_f\ parent_n\ =\ this_n$
  where $this_n$        $= mknode\ [$
                            $pp\ =\ $“(” $+\!\!+\ expr_n.pp\ +\!\!+$ “ $+_a$ ” $+\!\!+\ expr_n.pp\ +\!\!+$ “)”
                          $]\ \triangleright\ forwards_n$
        $expr_n$        $= expr_f\ this_n$
        $expr'_n$       $= expr'_f\ this_n$
        $forwards_n$    $= forwards_f\ parent_n$
        $forwards_f$    $= parseEnv\ [($“\$a”$, expr_f),\ ($“\$b”$, expr'_f)\ ]$
                            ( “ if $size(\$a)\ =\ size(\$b)$ then
                                $mkArray\ (\ size\ (\$a)\ ,\ \lambda\ i\ \rightarrow\ \$a[i]\ +\ \$b[i]\ )$
                                else $halt$ “adding arrays of different sizes” ” )
```

It forwards to a construct which checks, at run time, that the arrays have the same size, and if they do, uses the *mkArray* intention to create a new array whose elements are obtained by pointwise addition of the array operands. There are two interesting properties of the forwards–to construction. First, we show that we can use the generic subscript operation instead of using the array subscript operator. Since a is of array type (otherwise *plus* would not have forwarded to this construct) the generic subscript operator will correctly resolve the operator overloading. Second, we use the generic (+) operator. As long as the component types of the arrays are the same and also overload (+) this will succeed. If the component types do not overload (+) then that type will report the error. The *arrayAdd* intention does not have to check this.

3.2 Optimising array intentions

It could be argued that the above definition of *mkArray* is merely a poor substitute for the facilities offered by modern programming languages such as ML. After all, we could define our notion of arrays as an appropriate module, and hide the implementation details through an appropriate type system. That is true, and indeed we hope to construct a preprocessor that takes a module in PLX and produces the intention definitions in the previous section.

The true advantage of implementing arrays as intentions only becomes apparent when we consider our goal of increasing the efficiency of the programs written using the array intentions. The optimisation we will describe in this section lifts loop–invariant expressions from the implicit loop enclosing the initialisation function. For example, an optimising *mkArray* would transform itself from

$$mkArray\ (\,100,\ \lambda\, i\ \rightarrow\ mkArray\ (\,200,\ \lambda\, j\ \rightarrow\ j\,*\,j\ +\ i\,*\,i\,)\,)$$

to the following form in which the multiplication expressions are lifted so that they are not repeatedly, and needlessly, re–evaluated:

```
let t_1 := mkArray ( 200, λ j → j * j )
in mkArray ( 100, λ i →
                   let t_2 := i * i
                   in mkArray ( 200, λ j → t_1 [ j ] + t_2 )
                   end )
end
```

Here we have lifted the multiplication $i\,*\,i$ from the inner *mkArray*, where it was originally re–evaluated for each index value of j, to a *let*-binding enclosing the inner *mkArray* where it is evaluated just once for each value of i. We have also lifted the multiplication $j\,*\,j$ out of the inner *mkArray* where it was re–evaluated for each value of index i. In lifting $j\,*j$, we have moved the expression outside the scope of the lambda expression which binds the free variable j. In such cases, the temporary variable we create is not a simple scalar like t_2 above, but is an array holding the values which will be used more than once so that they are not re-computed. To create these temporary arrays, we will need to keep track of the size and index variable of the *mkArray* intentions from which we are lifting the expression. This is required so that the reconstructed *mkArray* intentions in the *let*-binding which has the correct sizes and their lambda expressions bind the correct free variable in the lifted expression. We call this information a lifted expression's *mkArray context* and it is stored for each *mkArray* whose indexing variable appears as a free variable in the lifted expression. For example, in the example above, the *mkArray* context of the lifted expression $j\,*\,j$ contains the size (200) and binding variable j for the single *mkArray* that this expression is lifted from. This information is used to construct the *mkArray* on the right hand side of the t_1 variable binding.

It is worthwhile to consider a few further representative examples of these optimisations. We shall need to ensure that the optimisation is applied transitively, in the sense that it can lift subexpressions from expressions that have already been lifted themselves. An example of this phenomenon is provided by

```
mkArray (100,
λ i → mkArray (200,
               λ j → mkArray(300,
                             λ k → j * j + i * i)))
```

The optimised version of this expression is a subtle variation of the previous example:

```
let t₁ := mkArray ( 200, λ j → j * j )
in mkArray ( 100, λ i →
               let t₂ := i * i
               in mkArray ( 200, λ j →
                              let t₃ := t₁ [ j ] + t₂
                              in mkArray (300, λ k → t₃)
                              end )
               end )
end
```

Note how the right-hand side of t_3 is itself an optimised expression. When building the right-hand side of *let* bindings, we must use optimised versions of the relevant subexpressions.

Obviously the expressions that are lifted must be free of side-effects, but by itself that is not enough. Consider

```
let i := 0
in mkArray (5 , λ j → begin i := i + 1 ; i end )
end
```

The value of this expression is $[1, 2, 3, 4, 5]$. The following is a wrongly optimised version of the same expression:

```
let i := 0
    t₀ := (i + 1)
in mkArray (5 , λ j → begin i := t₀ ; i end )
end
```

Its value is $[1, 1, 1, 1, 1]$. Although the expression $i + 1$ is itself free of side effects, it is not correct to lift it because it contains a variable that is being assigned to. It follows that we shall have to keep track of which expressions are pure, and which expressions contain mutated variables.

The programmer could do these optimisations by hand and use the definition of *mkArray* in the previous section. Alternatively, we could throw away the above *mkArray* definition and construct a new one which performs the desired optimisations. A better solution is to reuse the above *mkArray* intention and define an additional *optimising* array creation intention called *mkArrayO* that performs these optimisations *at the mkArray level.* It will forward to a *let*-binding, like those above, whose declaration section contains the lifted expressions which would be needlessly re–evaluated and whose body contains array creation intentions of the original *mkArray* variety, re–written in an optimised context. Thus, the original AST will contain only occurrences of *mkArrayO*. When these intentions are queried for their Java byte code attribute, for example, the query will be forwarded from our new *mkArrayO* intention, to the optimised *let*-binding containing the original *mkArray* intentions. As before, these in turn will forward to the appropriate *let*-binding containing the vector operations implementing the *mkArray* in the new optimised context.

There are two concerns which drive the specification of *mkArrayO*. First, we want to lift expressions to the highest possible point in the AST, that is, out of as many occurrences of *mkArray* as possible, to avoid unnecessary re-computation. This is discussed in the following section, Section 3.2.1. Second, we must ensure that the behaviour of the optimised program is the same as the un–optimised program; this is discussed in Section 3.2.3.

3.2.1 Lifting expressions

Four attributes are used to ensure that the expressions are lifted as high as possible:

- *possLifts* is the list of expressions that may be lifted and their associated *mkArray* contexts. The type of this (synthesised) attribute is

  $[\,(\,Node,\ [(Node,\ Node)]\,)\,]$

  Here the first element of each pair is an expression which may be lifted and the second is its *mkArray* context. We record such context information for each *mkArray* that binds an identifier that occurs in the lifted expression. The context information is a pair containing the size and the index variable of the corresponding *mkArray*, both represented as nodes. For example, for the lifted expression $j*j$ in the example at the beginning of this section, the context information is $[(200, j)]$.

- *rewrites* is the list of expression nodes which will be lifted by some enclosing *mkArrayO* and the function which builds the *Node* which replaces it in the rewritten AST. Its type is $[(\ Node,\ SemTree\ )]$. It is passed down the tree with enclosing occurrences of *mkArrayO* adding the expressions they lift to the list they inherited. Intentions access this attribute to find what they rewrite to. At each *mkArrayO*, the attribute is also used to see what expressions have already been lifted by an enclosing *mkArrayO*. This ensures that expressions are lifted by the "highest" possible *mkArrayO* in the AST. All intentions, except for *mkArray* get their values for *rewrites* from their parent.

- rwt_f has type *SemTree*; it is the tree function that an intention rewrites to. If a node finds that it appears as the first element of a pair in *rewrites*, then it defines its rwt_f attribute as the second element of that pair. If it does not find itself in *rewrites*, then it defines rwt_f as its own intention definition function with the children function parameters instantiated as the rwt_f of its children.

- $this_f$ also has type $SemTree$. It is the original function with the optimised child functions that generates the intention node given its parent node. This attribute value is used in constructing the right hand side of the *let* declarations which the rewriting $mkArrayO$ constructs.

All base PLX and array intentions define these four attributes in a separate "lifting" aspect. Determining which expressions to lift, *possLifts*, is quite an interesting problem, and it is discussed separately in Section 3.2.3. Here we provide the definitions of the other three attributes listed above for the multiplication intention. They are similarly defined for the other intentions.

```
aspect lifting : multiply $expr_n$ $expr'_n$ $parent_n$ $this_n$
  = [ $this_f$     = multiply ($expr_n.rwt_f$) ($expr'_n.rwt_f$),
      rewrites = $parent_n$.rewrites,
      $rwt_f$      = case lookup $this_n$ ($this_n$.rewrites) of
                       Just f  → f
                       Nothing → multiply ($expr_n.rwt_f$) ($expr'_n.rwt_f$)
    ]
```

All three of these attributes could be automatically defined using a slightly more advanced version of the defaulting mechanism presented in defined Section 2.4.3. We expose them here in a particular instance for expository reasons. It is interesting to examine the definition of $this_f$ closely: in the right-hand side, we create a (function that creates a) multiplication node with the *optimised* descendants $expr_n.rwt_f$ and $expr'_n.rwt_f$. Had we chosen to use the more obvious $expr_n.this_f$ and $expr'_n.this_f$ instead, the optimisations would not be applied to right-hand sides of *let*-bindings. In the introduction to this section, we called this effect the 'transitive application' of optimisations.

We can now define the optimising intention $mkArrayO$. It defines the *typei* attribute as before, but gets a value for the pretty print *pp* attribute from the *let*-binding it forwards to. Thus, the *pp* attribute gives us the optimised *mkArray* code. Its definition is as follows:

```
mkArrayO :: SemTree → SemTree → Node → Node
mkArrayO $size_f$ $init_f$ $parent_n$ = $this_n$
  where $this_n$    = mknode [
                     typei = arrayType ($init_n$.typei.returnType) ,
                     $this_f$ = mkArrayO $size_n.this_f$ $init_n.this_f$,
                     possLifts = mkAPossLifts,
                     rewrites = mkARewrites ++ $parent_n$.rewrites ,
                     $rwt_f$ = $forward_f$
                   ] ▷ ($forward_f$ $parent_n$)
        $size_n$    = $size_f$ $this_n$
        $init_n$    = $init_f$ $this_n$
        $forward_f$ = parseEnv [ ("$size", $size_n.rwt_f$), ("$init", $init_n.rwt_f$),
                               ("$letdcls", letdcls) ]
                      " let $letdcls in mkArray ( $size, $init ) end "
```

In this definition, *mkArrayO* forwards to a *let*-binding whose declarations contain the lifted expressions and whose body is the non–optimising *mkArray*. The children of that second *mkArray* have rewritten themselves by replacing their lifted expressions with the temporary variables declared in this *let*. These rewritten versions are found in the rwt_f attribute on $size_n$ and $init_n$. The instances of rwt_f are computed using the *rewrites* attribute values passed down the AST by the enclosing *mkArrayO* intentions.

Note that *mkArrayO* also defines the four new attributes $this_f$, rwt_f, *possLifts* and *rewrites*. The definition of $this_f$ is as expected: it uses *mkArrayO* and the optimised child functions $size_n.this_f$ and $init_n.this_f$ to make a function of type *SemTree*.

The intention *mkArrayO* is special in that it defines rwt_f, the optimised version of itself, to be the same as what it forwards to. This is because *mkArrayO* is *self optimising*; that is, it determines the form of its own optimisation. Most other intentions do not determine their optimised form. Instead they query the *rewrites* attribute to find out what, if anything, a *mkArrayO* has decided they should optimise themselves to.

To complete the definition of the intention *mkArrayO* we shall further elaborate the computation of

- its possible lifts, *mkAPossLifts*. This is done in Section 3.2.3.
- the rewrites made by this *mkArrayO*, that is *mkARewrites* and
- the declarations for the *let*, *letdcls*. These are both discussed in the following Section 3.2.2.

3.2.2 Creating let declarations and rewrites

The expressions that *mkArrayO* will lift will become components of right hand sides of *let* declarations in *letdcls*. These expressions are contained in the possible lifts attribute $this_n.possLifts$, but *mkArrayO* will only lift those expressions *expr* which satisfy the following criteria:

- *expr* should not contain the indexing variable of the initialisation function as a free variable
- *expr* has not already been lifted by an enclosing *mkArrayO* and does not contain an already lifted expression. The list of already lifted expressions is computed by the expression $map\,fst(parent_n.rewrites)$ on *mkArrayO*. This test ensures that expressions are lifted "as high as possible."
- *expr* is not a component of another expression in $this_n.possLifts$. This ensures that the largest possible expressions are lifted.

A simple predicate can be written to encode these criteria, which is then used by the *mkArrayO* intention to filter $this_n.possLifts$ to create the list *toLift*. From this list, we create four lists which will be used to define *letdcls* and *mkARewrites*. Their respective roles are summarised below:

- *tempVars* is the list of new temporary variables, one for each expression in *toLift*. These are terminal symbols and thus have type *Terminal*. They will become children to the *genRef* and *genLetDcl* intentions when used in *dclLhss* and *rwts*.

- *dclLhss* is the list of declarations for the left hand sides of the *let*-bindings and is created by mapping the generic *let* declaration intention function *genLetDcl* over the list of temporary variables. Put formally, we have $dclLhss = map\ genLetDcl\ tempVars$. The type of *dclLhss* is $[\ SemTree\ ]$.

- *dclRhss* contains the declaration's right hand sides and also has type $[\ SemTree\ ]$. It is constructed by enclosing the lifted expressions in an appropriate *mkArray* for each element of its *mkArray* context. Recall how the lifted expression $j * j$ is wrapped in the appropriate *mkArray* in our introductory example at the beginning of this section. The *dclRhss* list is created by mapping the *mkRhss* function over the list of lifted expressions and their contexts as follows:

  $dclRhss = map\ mkRhs\ toLift$

  This function takes an expression and its context as an ordered pair, and for each element in its context, wraps the expression in a new *mkArrayO* constructed using the size and index variable information from the array context.

  ```
  $mkRhs :: (Node, [(Node, Node)]) \to SemTree$
  $mkRhs\ expr_n\ [\,] = expr_n.this_f$
  $mkRhs\ expr_n\ ((size_n, index_n) : cs)$
    $= mkArrayO\ (\ size_n.this_f\ )$
                  $(\ lambda\ [\ index_n.this_f\ ]$
                          $(mkRhs\ expr_n\ cs)\ )$

  $mkRhs :: (Node\ ,\ [(Node, Node)]) \to SemTree$
  $mkRhs\ expr_n\ [\,] = expr_n.this_f$
  $mkRhs\ expr_n\ ((size_n, index_n) : cs)$
    $= parseEnv\ [\ (\$size,\ size_n.this_f), (\$index,\ index_n.this_f),$
                   $(\$rest,\ mkRhs\ expr_n\ cs)\ ]$
                 $``mkArray\ \$size\ (\lambda\ \$index \to \$rest"\ )$
  ```

- $exprRwt_{fs}$ contains the expression functions which the corresponding lifted expressions will rewrite to. It also has type $[\ SemTree\ ]$. For each index variable *index* in an expressions array context, we enclose the temporary variable from *tempVars* in a a generic subscript intention using *index*.

  ```
  $exprRwt_{fs} = zipWith\ (\ \lambda\,(expr, contexts)\ term \to$
                              $mkRwt_f\ term\ (reverse\ contexts))$
                           $toLift\ tempVars$
  $mkRwt_f :: Node \to [Context] \to SemTree$
  ```

$$
\begin{array}{l}
\mathit{mkRwt}_f\ \mathit{terminal}\ [\,] \;=\; \mathit{genRef}\ \mathit{terminal} \\
\mathit{mkRwt}_f\ \mathit{terminal}\ ((\mathit{size}, \mathit{index}) : \mathit{cs}) \\
\quad = \; \mathit{subscript}\ (\mathit{mkRwt}_f\ \mathit{terminal}\ \mathit{cs})\ (\mathit{index}.\mathit{this}_f)
\end{array}
$$

$$
\begin{array}{l}
\mathit{mkRwt}_f\ \mathit{terminal}\ ((\mathit{size}, \mathit{index}) : \mathit{cs}) \\
\quad = \; \mathit{parseEnv}\ [\ (\$\mathit{array},\ \mathit{mkRwt}_f\ \mathit{terminal}\ \mathit{cs}),\ (\$\mathit{index}, \mathit{index}.\mathit{this}_f) \\
\text{“ \$array [\$index] “}
\end{array}
$$

To compute *mkARewrites* we simply zip together the expressions to lift with the expression functions they will rewrite to.

$$\mathit{mkARewrites} \;=\; \mathit{zipWith}\ (\lambda\ (\mathit{expr}, \mathit{contexts})\ b\ \to\ (\mathit{expr}, b))\ \mathit{toLift}\ \mathit{exprRwt}_{fs}$$

With the left and right hand sides of the *let*-bindings and the declaration binding intention function *letDclBinding* we can create the list of *let*-bindings *letdcls*. The intention *letDclBinding* has two children, the first is a *genLetDcl* and the second is its initialising expression. The list *letdcls* is created as follows:

$$\mathit{letdcls} \;=\; \mathit{zipWith}\ \mathit{letDclBinding}\ \mathit{dclLhss}\ \mathit{dclRhss}$$

These are the declarations are used in the *let*-binding that the *mkArrayO* forwards to which we saw above.

3.2.3 Selecting expressions to lift

To ensure that the optimised program has the same behaviour as the original, we only lift expressions that are pure (free of assignments). We also do not lift expressions which may terminate improperly from a conditional or other expression which does not ensure it will be executed at least once. In the case of the conditional *if*, we only lift an expression that might abort if it appears in the condition or in both conditional branches. We must also ensure that we do not lift an expression to a position where its free variables would be out of the scope their declarations.

To ensure that these conditions are met, the defining functions of *possLifts* on different intentions will reference the following attributes:

- *pure*, which was discussed in Section 2.5.
- *okTermination* also has type *Bool* and is true on all intentions which can be guaranteed to terminate normally.
- *freeVars* has type [*Node*] and it lists declaration nodes for the free variables which are components of an intention.
- *assignedDcls* is the list of declaration nodes of variables which are assigned to in the program.

Normal Termination The attribute *okTermination* is defined in the same manner as *pure* and is also has circular definitions. Most intentions are guaranteed to terminate normally, and thus assign *True* to *okTermination* if their children are also guaranteed to terminate normally. Excluded from this group are while loops, which may not terminate; vector and array subscripts, which may have bound errors; and division since division by 0 causes a run time error. Function references are again a special case. Function references and declarations define *okTermination* in the same manner in which they define *pure* and are thus circular.

Free variables We will need to know which free variables appear explicitly inside an expression. The declaration nodes of these variables are kept in the attribute *freeVars*. Variable references define their value of *freeVars* to the singleton list containing their declaration. Most other intentions define *freeVars* by simply collecting the *freeVars* of their children. As expected, *let*-bindings and lambda expressions collect *freeVars* from their children, but remove those variable declarations they introduce in defining *freeVars*.

Assigned to declarations We have two occasions where we need to know the declaration nodes of variables which may be assigned to. We saw the first in the definitions of the *pure* and *okTermination* attributes. The second is discussed below in the definition of *possLifts*.

Three attributes are used. *assignedDclsSyn* contains the list of declaration nodes of all the variables assigned in the subtree below it. On intentions other than *assign* this attribute is just the union of the *assignedDclsSyn* values of its children. *assignedDcls* contains the list of declaration nodes of all the variables assigned in a program. The root node in the AST assigns its value of *assignedDclsSyn* to its value of *assignedDcls*. This value is passed down the tree in the same manner in which *env* is. *assignableDcls* contains the list of declaration nodes which will be assigned to if the intention appears on the left hand side of an assignment intention. This is needed since these left hand sides may be more complex that simple variable references. For example, the vector subscript operator defines *assignableDcls* to be the *assignableDcls* value of its first child and the variable reference intentions define *assignableDcls* to be the list containing their declaration. Thus, if a vector subscript appears on the left hand side of an assignment, when the *assign* intention computes its value for *assignedDcls* it will use the declaration of the vector being assigned to.

The *safety* aspect combines definitions of these attributes and a portion of it is shown below.[9]

```
aspect safety :
assign $var_n$ $expr_n$ $parent_n$ $this_n$
  = [ okTermination   = $var_n$.okTermination $\wedge$ $expr_n$.okTermination,
      freeVars        = $var_n$.freeVars $\cup$ $expr_n$.freeVars,
      assignedDclsSyn = $var_n$.assignableDcls $\cup$
```

[9]There is a small technical error in the definition of *okTermination* for *letDclForFunction*. It should be *false* if the function is recursive. This isn't taken into account in this definition.

```
                    $(var_n.assignedDclsSyn) \cup (expr_n.assignedDclsSyn)$ ]
$refToNumber\ dcl_n\ id_n\ parent_n\ this_n$
  = [ $okTermination$  = $True$,
      $freeVars$       = [ $dcl_n$ ],
      $assignableDcls$ = [ $dcl_n$ ] ]
$refToFunction\ dcl_n\ id_n\ parent_n\ this_n$
  = [ $okTermination$  = $dcl_n.okTermination$,
      $freeVars$       = [ $dcl_n$ ] ],
      $assignableDcls$ = [ $dcl_n$ ] ]
$vectorSub\ vec_n\ index_n\ parent_n\ this_n$
  = [ $okTermination$  = $vec_n.okTermination \wedge index_n.okTermination$,
      $freeVars$       = $vec_n.freeVars \cup index_n.freeVars$,
      $assignableDcls$ = $vec_n.assignableDcls$ ]
$letDclForFunction\ expr_n\ id_n\ parent_n\ this_n$
  = [ $okTermination$  = $(this_n \not\in this_n.assignedDcls)$
                           $\wedge\ expr_n.okTermination$ ,
      $freeVars$       = [$this_n$] ]
$lambdaDclForFunction\ expr_n\ id_n\ parent_n\ this_n$
  = [ $okTermination$  = $False$,
      $freeVars$       = [$this_n$] ]
```

Defining possible lifts Defining the possible lifts is straightforward given the attributes defined above; thus we provide just a few examples. For the *multiply* intention, we include its node in *possLifts* if its is pure, terminates normally and does not include as a free variable any variable which may be assigned to. If this test fails we just use the values of its children.

```
**aspect** $arrayExtras$ :
$multiply\ expr_n\ expr'_n\ parent_n\ this_n$
  = [ $possLifts$ = $expr_n.possLifts \cup expr'_n.possLifts \cup$
                    (**if** $this_n.pure \wedge this_n.okTermination \wedge$
                        $(this_n.freeVars \cap this_n.assignedDcls = \emptyset)$
                    **then** [ ($this_n$, []) ] **else** [ ] ) ]
```

We define *possLifts* for the *if* intention to include the *if* itself if it passes the same test as *multiply* above. We also include the possible lifts from the condition expression, and those from the *then* and *else* branch which terminate normally (*okTermination* is *true*). We also include those possible lifts on the *then* branch which do not terminate normally but which also appear in the *else* branch. The computation for the *else* branch itself is symmetric.

```
**aspect** $arrayExtras$ :
$ifExpr\ cond_n\ thenBody_n\ elseBody_n\ parent_n\ this_n$
  = [ $possLifts$ = $cond_n.possLifts$ ++
                    [$lift$ | $lift \leftarrow (thenBody_n.possLifts)$,
              ( (fst lift).okTermination ∨
               or (map (liftEq lift) (elseList_n.possLifts) ) ) ] ++
    [lift |  lift ← (elseList_n.possLifts),
              ( (expr lift).okTermination ∨
               or (map (liftEq lift) (thenList_n.possLifts) ) ) ] ++
    (if this_n.pure ∧ this_n.okTermination ∧
       (this_n.freeVars ∩ this_n.assignedDcls = ∅)
    then [ (this_n, []) ] else [] ) ]
```

In the case of the *let*-binding, we also filter out those which expressions whose free variables are not a subset of the free variables of the *let* declaration. That is, we only include in *possLifts* those elements of the declaration and body *possLifts* whose expression contains no free variables defined by the *let*. This is easily accomplished since the expressions in *possLifts* on the *let*'s children (*fst l*) are queried for their *freeVars* attribute and the subset test is performed between this value and the free variables on the let binding. Lambda expressions which do not occur as the initialising function of a *mkArray* are treated similarly. Those which do, define their possible lifts to be just those on the expression of the lambda. This way, we do not lift *mkArray* initialising functions, but we may lift expressions which contain the free variable bound by this lambda.

```
aspect arrayExtras:
let dclList_n exprList_n parent_n this_n
  = [ possLifts = ( filter ( λ l → ((fst l).freeVars) ⊆ this_n.freeVars )
                     ( dcl_l ist_n.possLifts ∪ exprList_n.possLifts ) ) ++
                  ( if this_n.pure ∧ this_n.okTermination
                    then [ ( this_n, [] ) ] else [] ) ]
```

Defining possible lifts for *mkArrayO* simply takes the possible lifts from its initialisation function and adds context information for it if the possibly lifted expression contains the index variable of its initialisation function ($index_n$) as a free variable. The attribute *bindingVarDcl* is defined only on lambda expression under a *mkArray* and contains the declaration of the single variable.

```
mkAPossLifts
  = map checkIndex (init_n.possLifts)
      where checkIndex l = if index_n ∈ ((fst l).freeVars)
                           then ((fst l), (size_n, index_n) : (snd l) )
                           else l}

index_n = init_n.bindingVarDcl
```

The above definitions appear in the **where** clause of *mkArrayO* intention defined above and not in a separate aspect like the others shown here.

4 Extending PLX with case statements

In the introduction, we mentioned the addition of a *case* intention and that it should not require us to alter the array intentions for these intentions to co–exist. Thus, all the work we have done above is intended to make the addition of the case intention a simple process which requires no knowledge of the array intentions.

We introduce three case statement intentions: *case*, *caseOption*, and *otherwise*. As we saw in the introduction, a case statement will forward to a *let*-binding which stores the value of the case expression in a temporary variable and whose body is an if–then–else construct built by the case option intention. The case intention takes two children: the expression and the case options as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{case} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node}} \\
\multicolumn{3}{l}{\mathit{case}\ \mathit{expr}_f\ \mathit{option}_f\ \mathit{parent}_n\ =\ \mathit{this}_n} \\
\quad \textbf{where}\ \mathit{this}_n & = & \mathit{mknode}\ [\ \mathit{caseTemp}_f\ =\ \mathit{genRef}\ \mathit{temp} \\
 & & \qquad\qquad\quad \mathit{pp}\ = \text{“}\mathit{case}\text{ ”} +\!\!+\ \mathit{expr}_n.\mathit{pp}\ +\!\!+\ \text{“ }\mathit{of}\text{ ”} +\!\!+ \\
 & & \qquad\qquad\qquad\quad \mathit{option}_n.\mathit{pp}\ ]\ \triangleright\ \mathit{forward}_n \\
\qquad\quad \mathit{expr}_n & = & \mathit{expr}_f\ \mathit{this}_n \\
\qquad\quad \mathit{option}_n & = & \mathit{option}_f\ \mathit{this}_n \\
\qquad\quad \mathit{forward}_n & = & \mathit{forward}_f\ \mathit{parent}_n \\
\qquad\quad \mathit{forward}_f & = & \mathit{parseEnv}\ [(\text{“}\$\mathit{tmp}\text{”}, \mathit{getLetDcl}\ \mathit{temp}),\ (\text{“}\$\mathit{expr}\text{”}, \mathit{expr}_f), \\
 & & \qquad\qquad (\text{“}\$\mathit{equivIf}\text{”},\ \mathit{option}_n.\mathit{caseFwd}_f\ )\ ] \\
 & & \qquad\qquad \text{“}\textbf{let}\ \$\mathit{tmp}\ :=\ \$\mathit{expr}\ \textbf{in}\ \$\mathit{equivIf}\text{ ”} \\
\qquad\quad \mathit{temp} & :: & \mathit{Terminal} \\
\qquad\quad \mathit{temp} & = & \mathit{newTempVar}\ ()
\end{array}
$$

It defines the pretty print attribute *pp* and the $\mathit{caseTemp}_f$ attribute; this is the *SemTree* function to build the generic reference to the temporary variable containing the value of the expression and it is used in the case options in constructing their equivalent if–then–else construct. We use the function *newTempVar*() to generate a *Terminal* called *temp* with a unique lexeme. The *let*-binding that the case forwards to is constructed from the temporary variable declaration and and case expression *expr* as expected as well as from the if–then–else construct retrieved from the $\mathit{caseFwd}_f$ attribute on the case option child option_n. As we saw in Section 2.4.2 the generic reference and declaration attributes use the *env* attribute to link references to declarations. Although it would require additional work and make the definition of *case* and *caseOption* less clear, we could use type type-specific reference and declaration intentions by querying $\mathit{expr}_n.\mathit{typei}$ for its *refSpecializer* and *genLetDclSpecializer* attribute and manually link the references to the temporary variable to its declaration.

The case option intention takes three children: the expression to compare to the case expression temporary variable, the expression to return if the comparison succeeds, and the next case option in the sequence. The last case option is the *otherwise* intention. *caseOption* and *otherwise* define the $\mathit{caseFwd}_f$ attribute which the *case* intention uses to build the *let*-binding it forwards to. These intentions are defined as follows:

$$\mathit{caseOption} :: \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{SemTree} \rightarrow \mathit{Node} \rightarrow \mathit{Node}$$

$$\begin{array}{l}
\mathit{caseOption}\ \mathit{cond}_f\ \mathit{body}_f\ \mathit{nextOpt}_f\ \mathit{parent}_n\ =\ \mathit{this}_n \\
\quad \mathbf{where}\ \mathit{this}_n \quad = \mathit{mknode}\ [\ \mathit{caseTemp}_f\ =\ \mathit{parent}_n.\mathit{caseTemp}_f, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{caseFwd}_f\ =\ \mathit{caseForward}_f \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pp}\ = \mathit{cond}_n.\mathit{pp}\ +\!\!+\ \text{“}\ \rightarrow\ \text{”}\ +\!\!+\ \mathit{body.n.pp}\ +\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{“ ; ”}\ +\!\!+\ \mathit{nextOpt}_f.\mathit{pp} \\
\qquad\qquad\qquad\quad]\ \rhd\ (\mathit{defaults}\ \mathit{this}_n\ [\mathit{cond}_n, \mathit{body}_n, \mathit{nextOpt}_n]\ \mathit{parent}_n\ ) \\
\qquad\qquad \mathit{cond}_n \quad = \mathit{cond}_f\ \mathit{this}_n \\
\qquad\qquad \mathit{body}_n \quad = \mathit{body}_f\ \mathit{this}_n \\
\qquad\qquad \mathit{nextOpt}_n = \mathit{nextOpt}_f\ \mathit{this}_n \\
\qquad\qquad \mathit{caseForward}_f\ = \mathit{parseEnv}\ [(\text{“}\$\mathit{tmp}\text{”}, \mathit{this}_n.\mathit{caseTemp}_f) \\
\qquad\qquad\qquad\qquad\qquad\quad (\text{“}\$\mathit{cond}\text{”}, \mathit{cond}_f), (\text{“}\$\mathit{body}\text{”}, \mathit{body}_f), \\
\qquad\qquad\qquad\qquad\qquad\quad (\text{“}\$\mathit{next}\text{”}, \mathit{nextOpt}_n.\mathit{caseFwd}_f)\ ] \\
\qquad\qquad\qquad\qquad\qquad\quad \text{“}\mathbf{if}\ \$\mathit{tmp}\ =\ \$\mathit{cond}\ \mathbf{then}\ \$\mathit{body}\ \mathbf{else}\ \$\mathit{next}\text{”}
\end{array}$$

$$\begin{array}{l}
\mathit{caseOtherwise}\ ::\ \mathit{SemTree}\ \rightarrow\ \mathit{Node}\ \rightarrow\ \mathit{Node} \\
\mathit{caseOtherwise}\ \mathit{body}_f\ \mathit{parent}_n\ =\ \mathit{this}_n \\
\quad \mathbf{where}\ \mathit{this}_n = \mathit{mknode}\ [\ \mathit{caseFwd}_f\ =\ \mathit{body}_f, \\
\qquad\qquad\qquad\qquad\qquad \mathit{pp}\ =\ \text{“}\mathit{otherwise}\ \rightarrow\ \text{”}\ +\!\!+\ \mathit{body}_n.\mathit{pp} \\
\qquad\qquad\qquad\qquad\ ]\ \rhd\ (\mathit{defaults}\ \mathit{this}_n\ [\ \mathit{body}_n\ ]\ \mathit{parent}_n\ )
\end{array}$$

Since the *case* intention forwards to a semantically equivalent *let*-binding with nested *if* intentions it does not need to provide definitions for the attributes we introduced above *mkArray*; *e.g. possLifts*, *rewrites*, and *pure*. All these attributes are defined by *let*-binding that case forwards to.

5 Discussion

5.1 Connections with R5.0

Our model simplifies some of the more complicated aspects of R5.0. In particular, "bidirectional links" are used there to allow tree nodes to access their parents for context information, and there is an associated mechanism for ensuring that this links are maintained correctly on forwarded nodes. In addition, intentions are handed information about whether it is a directly connected node (and if so, which one) that is requesting the information when a "question" is asked; this gives intentions that construct compound trees to forward to the ability to select the appropriate context to which to redirect this question. This mechanism leads to some rather complex restrictions to ensure that it is possible to know the "source" of a question if necessary. In contrast, our model circumvents this problem by representing trees as functions of their context; thus forwarding is handled by constructing compound trees out of these functions rather than concrete trees, and the correct context is always present when an attribute lookup is made.

For example, consider the for-into-while forwarding transformation from Section 2.2.1. In R5.0, the abstract syntax for a statement block containing the initialisation and the while loop would be explicitly constructed (it is intended that there will be a "quoting" mechanism to simplify this), and a handle to the while node would be kept. Questions from the body or the condition would then be directed to the while loop, while questions from the initialisation statement or anywhere else would be forwarded to the statement block. In our model, the tree to forward to would be constructed in a similar fashion; however attribute lookups from the body or condition for which it was necessary for them to know their context would happen as a result of an initial lookup that came via the while node above them in the newly constructed tree; thus they would be passed this while node as part of their context and would immediately know the correct node to direct their lookups to.

Another significant departure from the R5.0 approach is in the handling of variable references. Our model is not based on a structure editor, and thus for simplicity, we choose to identify these by variable name and bind references to declarations during reduction, although it would not be hard to use a GUID-based system or something similar as R5.0 does. Likewise, for convenience we have chosen to attach child trees to nodes positionally rather than with "tags".

As we commented earlier, our model omits various advanced features currently present in R5.0. Two of these are system support for "build targets", which allow forwarding to be parameterised by a global variable denoting the current target platform, and errors, which provide an exception mechanism for allowing errors in derived source to be caught by the forwarding intention and either suppressed or translated into more appropriate messages pertaining to the original source code. Both of these features could be implemented without fundamental changes. In particular, an exception mechanism would mesh well with build target support; if an intention had a choice of transformations for a particular build target, and some of those transformations were more optimal but not guaranteed to be compatible with other intentions, one of them could be chosen and an exception later raised if necessary. Because our model does not make use of updatable state, there would be no need for a heavyweight rollback mechanism to support this.

A currently undecided question in the design of R5.0 is the interplay between forwarding and defaulting. Specifically, if a default handler for a question is reached after following a chain of forwarding links, and itself needs to ask questions in order to provide an answer, then should these questions be directed to the beginning of the chain of forwarding links, or the end? Currently, our model chooses the latter option, but modification to the $\triangleright$ forwarding operator or the introduction of multiple alternatives within one implementation would allow this behaviour to be modified.

5.2 Modularity

Achieving a high degree of modularity in programming language design and implementation is a prerequisite for *intentional programming.* Throughout the design of PLX and the *mkArray* intention are several distinct features which increase the modularity of the language. In this section we revisit some of these features to emphasise their importance and since their significance to increasing the modularity of language design can get lost in the details of the discussion above.

Forwarding Forwarding plays a crucial role in both our model and the R5.0 implementation.

- *reuse of previous intentions.* One often states that one intention depends (forwards-to) another to define some of its attributes. This means that we don't have to duplicate the work that went into creating the forwarded-to intention. We saw this in the *for* loop intention forwarding-to the statement list containing and initialising assignment and *while* loop.
- *links from references to declarations.* The scoping rules are part of the base language. Since we would like our sets of intentions to work with more than one base language, the variable references which we define in our intentions should not be aware of the mechanism used to link references to their declarations. The generic reference intention *genRef* enforces the scoping rules of the base language and performs the linking of references to declarations. It also forwards all other attribute queries to a type-specific reference intention. This allows the *arrayRef* intention to simply assume it is linked to its declaration in which ever base language it is used.
- *operator overloading.* As mentioned in Section 2.4.1, the specific manner in which we use the "specializer" attributes to implement operator overloading enables additional types to overload a generic operator without making any changes to the generic operator itself or to the other intentions which overload it. This type of modularity is critical if one hopes to create languages by additively combining intentions.

Aspects It almost goes without saying that the use of aspects is essential in achieving a modular definition of a language. We often need to write definitions for a new set of attributes for many existing intentions. By placing these definition in a new aspect we do not have to modify the original intentions to add these attribute definitions.

New types Additional types can easily be added into the system. Determining type equality for new types does not require the modification of a global type equality algorithm. Such an algorithm is in a sense distributed across the intentions in the form of attribute definitions.

There is a distinction to be made between the features of this model and how these features are used. Forwarding and aspects can be used in a modular way but *how* they are used is as important as the decision to use them in the first place. It is possible to use forwarding and aspects in a non-modular way as we saw in the non-modular definition of the generic *plus* operator in Section 2.4.1.

5.3 Future Work

There are several current research efforts that are beginning to show results but are not yet incorporated into our model of reduction. We briefly describe some of them here.

- Kevin Backhouse is investigating how the circularity and completeness tests for standard attribute grammars and higher order attribute grammars can be applied to our model. The circularity test will be used to identify circular definitions in language specifications. Some circularities, like those in the definition of the *pure* attribute, are desirable, but some are the result of errors in the language specifications. Identifying these unwanted circularities can be quite difficult and thus have such a test will be important in a production IP system.

- Some members of the Oxford team have been investigating how transformations expressed as rewrite-rules can be incorporated into attribute grammars and our model. It seems clear that it can be done but some details remain to be sorted out. Being able to embed rewrite-rule based transformations will enable the work of Ivan Sanabria-Piretti and Ganesh Sittampalam to be more directly applied to our model.

- Ivan Sanabria-Piretti's research is concerned with transforming domain-specific abstract data types (ADTs) into one of several possible concrete representations in the context of imperative languages. A data-refinement process based on term rewriting takes place on the abstract specification and produces a resulting program given in terms of the desired concrete implementation of the ADTs. Many optimising transformation are also specified as rewrite rules so that the resulting concrete implementation is efficient. Integrating this work into our model should be possible once a clearer idea of how term rewriting can be deployed in it.

- Ganesh Sittampalam's work has shown how sophisticated higher-order pattern matching algorithms can be used to apply complex optimisations to programs by rewriting. Such optimisations will be important in allowing high-level intentions to be implemented efficiently. As with Sanabria-Piretti's work we hope to be able to integrate the higher-order rewriting rules into our model.

- David Lacey, Oege de Moor, and Eric Van Wyk have been investigating the use of temporal logic as a concise and formal method for specifying control flow conditions. Recall that in Section 2.5 we defined a function declaration to be *pure* if its function body was pure and the function variable was not assigned to at any point in the program. This is overly conservative. A more accurate test for checking when an expression that contains a function reference can be lifted from program point a to program point b would check that there are no assignments to the function variable on any paths from a to b in the control flow graph of the program. Such conditions can be simply and formally stated as temporal logic formulae over the control flow graph. We could therefore define more accurate tests of control flow conditions in a concise and formal notation and thus simplify the process of defining intention optimisations.

A Haskell Primer

The meaning of some of the features and functions of Haskell [PJHA⁺99] are not immediately obvious and they are presented here for readers unfamiliar with the language. Bird, in [Bir98], provides a nice introduction to functional programming using Haskell which readers should consult for a much more complete treatment of the language.

1. Lists are commonly used in Haskell and in our intention definitions. "[ ]" denotes the empty list, and "$(b\ :bs)$" denotes the list constructed by appending b to the front of the list bs. (:) is called the *cons* operator. List concatenation is performed by the operator $(+\!\!+)$. Since strings are lists of characters this is also string concatenation.

2. Function application is written without parenthesis and are curried. Consider the function $add3$ which sums 3 numbers.

   $add3 :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$
   $add3\ a\ b\ c\ =\ a\ +\ b\ +\ c$

   An application of $add3$ is written $add3\ \ a\ \ b\ \ c$ instead of $add3\ (a, b, c)$. Since functions are *curried* we do not need to provide them with all of their parameters at once. To illustrate this consider the phrase $add3\ \ 2\ \ 3$ which evaluates to a function of type $\mathit{Int} \rightarrow \mathit{Int}$ that has already instantiated parameters a and b as 2 and 3 respectively. When this unary function is applied to a number, it returns its value plus 5.

3. Function definition via pattern matching allows one to write several function definitions; the right hand side of each contains a pattern matching a different possible form of the input parameters.

   In the definition of $(\oplus)$ in Section 2.1.2 shown below, there are two definitions of $(\oplus)$. In both the name *node* matches any possible form of the first parameter but second parameter, a list of $\mathit{Node} \rightarrow \mathit{Node}$ attribute definition functions, is matched against the patterns [ ] and $(\mathit{attrDef} : \mathit{attrDefs})$. If the second parameter is the empty list, it matches [ ] and the value of *node* is returned as the value of the function. If the list is not empty, *attrDef* is instantiated to the head of the list, and *attrDefs* to the tail. These values can then be referenced by the names in the pattern in computing the value of the function.

   $(\oplus) :: \mathit{Node} \rightarrow [\mathit{Node} \rightarrow \mathit{Node}] \rightarrow \mathit{Node}$
   $\mathit{node} \oplus [\ ] = \mathit{node}$
   $\mathit{node} \oplus (\mathit{attrDef} : \mathit{attrDefs})\ =\ \mathit{attrDef}\ (\mathit{node} \oplus \mathit{attrDefs})$

4. The function *zipWith* has type $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma]$. It takes a binary function $(\oplus)$ and two lists x and y to create a third list z so that $z_i = x_i \oplus y_i$.

5. The function *filter* has type $(\alpha \rightarrow \mathit{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$. It takes a predicate p and a list xs applies it to each element in xs. Those elements of $x \in xs$ for which $p\ x = \mathit{True}$ are returned in the output list.

References

[Bir98] R. S Bird. *Introduction to Functional Programming in Haskell.* International Series in Computer Science. Prentice Hall, 1998.

[CM97] L.M. Chirca and D.F. Martin. An order-algebraic definition of knuthian semantics. *Mathematical Systems Theory*, 13(1), 1997.

[dMPJVW99] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect oriented compilers. In K. Czarnacki and U. Eisenecker, editors, *First International Symposium on Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 121–133, 1999.

[Far86] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *ACM SIGPLAN Notices*, 21(7), 1986.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, 1992.

[Hed99] G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.

[Joh87] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.

[Jon90] L.G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–46, 1990.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997.

[MLAV99] M. Mernik, M. Lenic, E. Avdicausevic, and Z. Viljem. Multiple attribute grammar inheritance. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.

[PJHA+99] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98. Available at URL: `http://www.haskell.org`, February 1999.

[Sim96] C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1 workshop. Available at URL `http://www.research.microsoft.com/research/ip/`, 1996.

[Sim99] C. Simonyi. The future is intentional. *Computer*, May 1999.

[SV91] D. Swierstra and H. Vogt. Higher-order attribute grammars. In H. Albas and B. Melichar, editors, *International Summer School on Attribute Grammars Applications and Systens: SAGA*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer-Verlag, 1991.

[VWdMBK02] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, 2002. To appear.