Intentional Programming

British Computer Society

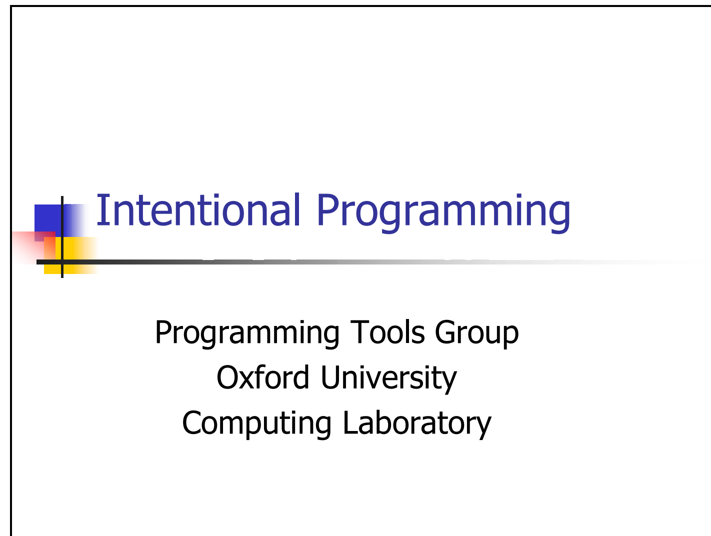
Advanced Programming Specialist Group

<table>
<tr><td>Speaker</td><td>Dr Oege de Moor, Oxford University</td></tr>
<tr><td>Time, Date</td><td>6 pm Thursday 8th February 2001</td></tr>
<tr><td>Summary</td><td>Programming languages and programming tasks are rarely a perfect fit: often a program could be clarified greatly 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 is implemented in a highly modular fashion, that cost might be lower.

To achieve such modularisation is the goal of intentional programming; to emphasise the fact that the language features can be tailored to the programmer's wishes, they are called intentions.

In this talk, we introduce the basic concepts of intentional programming, and relate them to earlier attempts to achieve the same goals, in particular in the literature on attribute grammars.</td></tr>
<tr><td>Venue</td><td>Sun Microsystems, Regis House, 45 King William Street, London EC4. Nearest Underground stations, Bank, Cannon Street</td></tr>
</table>

Slide 1



I would like to thank Professor Florentin for the invitation to give this talk. The work that I am about to describe is an attempt to distil and synthesise the main technical ideas that underpin Charles Simonyi's vision of *intentional programming*. The research was carried out by a number of people, all members of the Programming Tools Group at Oxford University Computing Laboratory. In particular, today's talk owes a lot to the efforts of Eric van Wyk, a research officer whose work at Oxford is supported by Microsoft. Kevin Backhouse, Ivan Sanabria-Piretti, and Ganesh Sittampalam also contributed to the content. I will only be able to give you a bird's eye view of what our research team is doing: check out our web site at Oxford for seminars, software and research papers.

Programming languages and programming tasks are rarely a perfect fit. It is common practice to tailor a language to an application domain by designing an *application framework*. Although this became a fashionable activity only recently, the creation of such *special application languages* was an explicitly stated aim of Simula 67. Reading the preface of Simula's language definition (published in 1968) is an inspirational experience: clearly its designers were way ahead of their time.

Today perhaps the most widely known example of a framework is provided by *Microsoft's Foundation Classes*, MFC for short. The code for MFC shows that even a large language such as C++ is creaking at the seams when tailored to specific tasks: macros abound whenever C++ is not up to the task in hand. Even worse, a lot of the work is done through *application wizards* that simply generate boilerplate code that is common to many applications. Such application wizards are run only once, when a new project is started, and they cannot be invoked again when revising the original code. While some of these problems may be due to bad engineering practice by the designers of MFC, they indicate that Bjarne Stroustrup was not successful in his aim of eradicating the use of the C pre-processor. Making the pre-processor obsolete was one of his reasons for designing C++ in the first place.

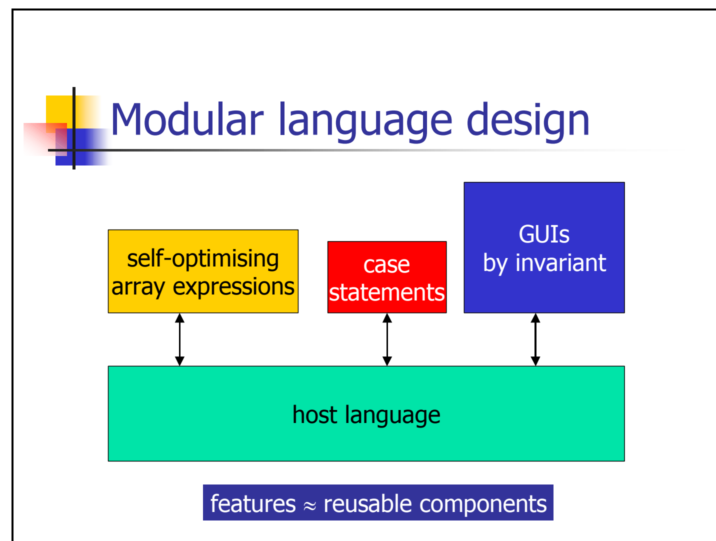
Why is it not possible to integrate the features of MFC into C++? The main argument is one of economy: extending a language definition and its compiler is a major effort. If it were easy to build language implementations from plug-and-play components, it might be less costly to tailor a language to a specific application domain. Such lightweight, modular language implementation is the goal of *intentional programming*.

The vision of intentional programming is due to Charles Simonyi, the founder of Microsoft's applications division, and the author of Microsoft Word. His aim is that programmers can express their *intentions* explicitly in the code, rather than implicitly via inadequate language features. The intentions are thus the plug-and-play components of a programming language implementation. During the next fifty minutes, I hope to show you how that vision can be realised.

I should hasten to say that there is much more to Simonyi's vision than merely a method of language implementation. In a number of papers and lectures he has argued eloquently that other aspects of the programming process must also be improved to allow a more direct expression of a programmer's intent. For instance, one needs an ergonomically designed structure editor, complete freedom of notation, a debugger that provides output in domain-specific terms (so it could, for example, draw a graph in lieu of dumping a pointer structure in hexadecimal form), and so on. Simonyi and his colleagues at Microsoft have been building a sophisticated system with these desirable characteristics since the early nineties. I warmly agree that all these other aspects are very important, but I shall ignore them in today's talk, and concentrate exclusively on modular language implementation.

When I accepted the invitation of this talk, Prof. Florentin mentioned that it would be nice if I could point out how Intentional Programming differs from previous attempts to achieve the same objectives. Indeed, this has been part of our research method. Inspired by the requirements and ideas of Simonyi's team, we have been able to select and combine working solutions from many pioneering works of others. In particular, we have learnt a lot from previous work on transformational programming and from the attribute grammar community.

Slide 2

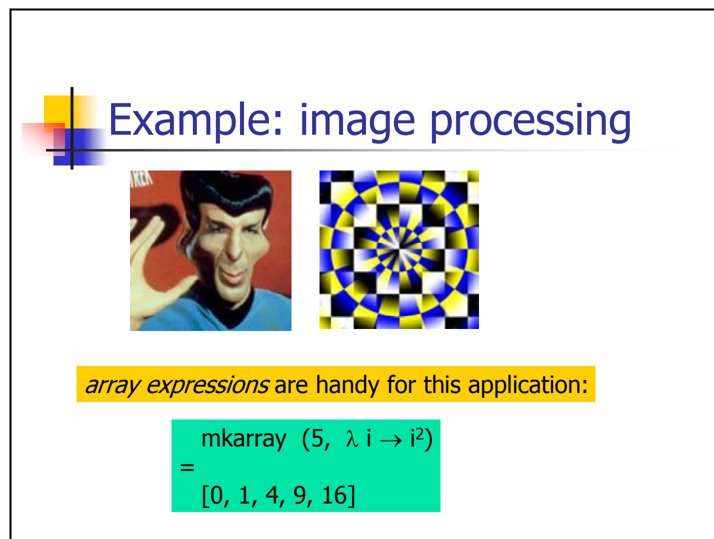


This diagram illustrates our goal. We wish to start with a host language, which is a programming language much like those in common use today. However, the host language will typically be somewhat minimalist, leaving out all syntactic sugar. It might for instance provide only a *while*-loop, because *for*-loops and *repeat-untils* can be easily phrased in terms of *while*.

That Spartan host language can then be extended by a rich set of features, and each of these features is a re-usable component. The host language is tailored to a new programming task by importing an appropriate set of such components. Of course it is important that the components are independent of each other: achieving such independence is the main technical challenge, especially if we wish to apply application-specific optimisations whenever new features are used in client code.

On this slide I have given three examples of the kind of components one might wish to describe in this way: a feature for describing GUIs by stating invariant relationships between GUI elements, a general *case* statement, and self-optimising array expressions. It is the arrays example that will guide the remainder of this talk.

Slide 3



Last year, I was working with some graphics researchers on ways to program graphical animations quickly. Our programs did a lot of image processing, and much of the work went into initialising appropriate array structures. Most of the necessary speed came from careful optimisation of these array initialisers.

Our programming language only provided vectors (essentially just pointers), along with a *malloc*-like primitive to allocate space. Clearly it would be very useful to have a new type of arrays, along with a new form of expression that creates values of this type. This new expression form, called *mkarray*, is illustrated on the slide: it takes two parameters, namely the size (here 5) and the initialiser, which is a function from indices to values.

The example shows how index *i* gets mapped to the square of *i*. The value of *mkarray* $(5, \lambda i \rightarrow i^2)$ is thus an array of 5 elements, where the ith element is i^2. The notation $(\lambda i \rightarrow i^2)$ is borrowed from functional programming: it denotes an anonymous function that takes a parameter named *i*, and it returns i^2.

Slide 4

Array expressions

new feature: mkarray (size, λ i → E(i))

equivalent in host language:

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

To make this precise, here is the new feature we wish to introduce, along with its equivalent in the host language.

I shall be using a host language where all statements are expressions: this makes it easier to formulate the transformations I wish to discuss. It is not necessary that the host language has this property, and it is perfectly possible to apply these ideas to extending a more common host language such as C. It would however be rather awkward to illustrate the same idea in C, because in C, variable declarations cannot be part of expressions.

The host program starts by declaring (and initialising) two local variables, named *v* and *i*. Here *v* is a vector (a pointer to a block of memory) of the given *size*. The other variable *i* is the index variable used in the initialising loop. The body of this local block consists of a *while* statement that assigns each array element in turn. Finally, we return the vector *v* as a result of the computation.

Now you might say that we could just introduce a procedure for *mkarray*: after all, it is merely shorthand for allocating the space for an array, and then filling in the entries. But there is more to it than that: we also wish the implementation to automatically apply the optimisations that were originally performed by hand.

Slide 5

Desired optimisations

```
mkarray (N,   λ i →
                  mkarray (K, λ j →
                                   f(i) + g(j)  )  )
```

N*K evaluations of f and g

```
let x := mkarray (K, λ j → g(j))
in mkarray(N,    λ i →
                 let y := f(i)
                 in mkarray (K,   λ j → y + x[j])
                 end)
end
```

N evaluations of f,
K evaluations of g

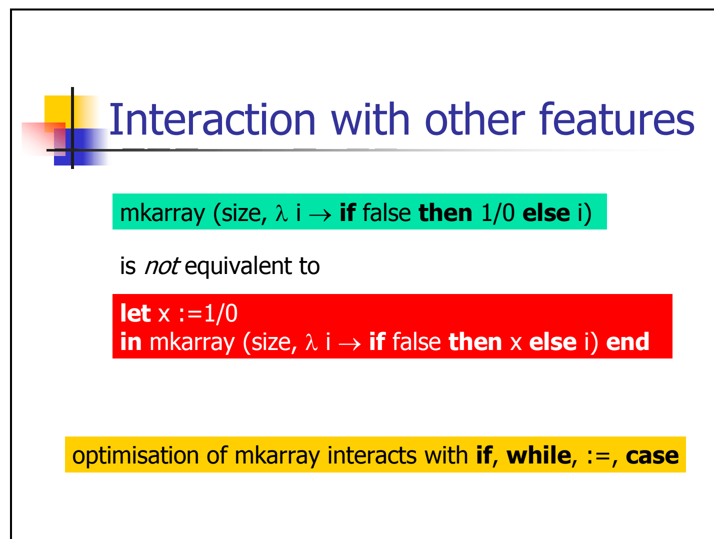
Here is an example to illustrate the optimisations I have in mind.

Consider the nested use of *mkarray* in the top box. It consists of an array of *N* elements; each element is itself an array of size *K*. At position (*i,j*) in this two-dimensional array, we store the value *f(i)+g(j)*. When expanded into the host language equivalent, this will give us a nested loop, where the inner loop body is executed *N*K* times. The functions *f* and *g* are therefore each evaluated *N*K* times. When *f* and *g* are non-trivial, this is clearly wasteful, because *f(i)* is the same for every value of *j*, and *g(j)* is unchanged for every value of *i*.

By contrast, let us now look at the more complex expression given below. We start by caching the value of *g* for every index *j*. Inside the outer *mkarray*, we cache the value of *f(i)* for one particular value of *i*. In the inner array body, we can now compute the value of *f(i)+g(j)* by computing the sum *y+x[j]*. As a result, there are only *N* evaluations of *f*, and *K* evaluations of *g*. The optimised program is therefore significantly faster.

In the image processing application I told you about, it is not uncommon to have five or six nested array expressions. Systematically applying these optimisations by hand is tedious and error-prone: we want them to be applied automatically.

Slide 6

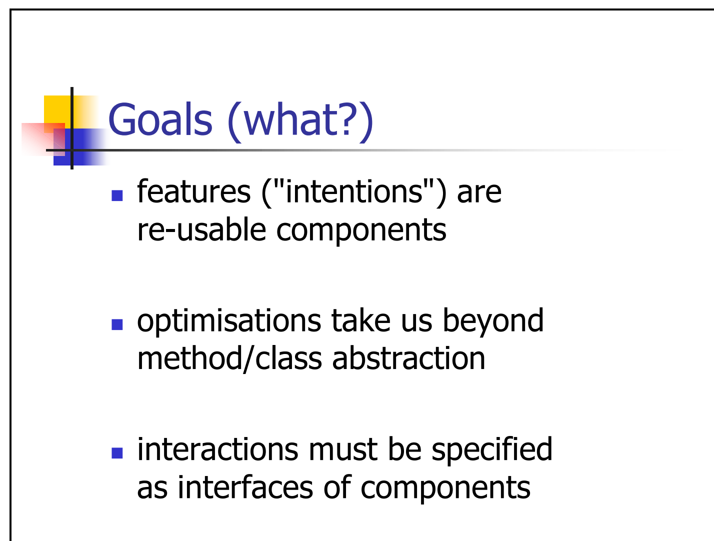


Part of the difficulty in applying these optimisations automatically is the interaction between our new *mkarray* expressions and other language features.

Here is an artificial example where it is definitely *not* correct to hoist an index-invariant expression out of the array construction. In the upper, unoptimised use of *mkarray*, 1/0 is never evaluated. By contrast, in the lower version, 1/0 is evaluated before the array computation, and therefore a run-time exception will occur. The root of the problem is the conditional expression inside *mkarray*. Although this example was especially concocted to illustrate the problem, the same phenomenon does of course happen in practice.

Similar examples show that we have to be careful about optimising array expressions in the presence of *while* and assignment (:=) statements. Unfortunately we even have to cater for features (such as *case*) that may be imported from other intention packages, unknown to us when implementing *mkarray*.

Slide 7



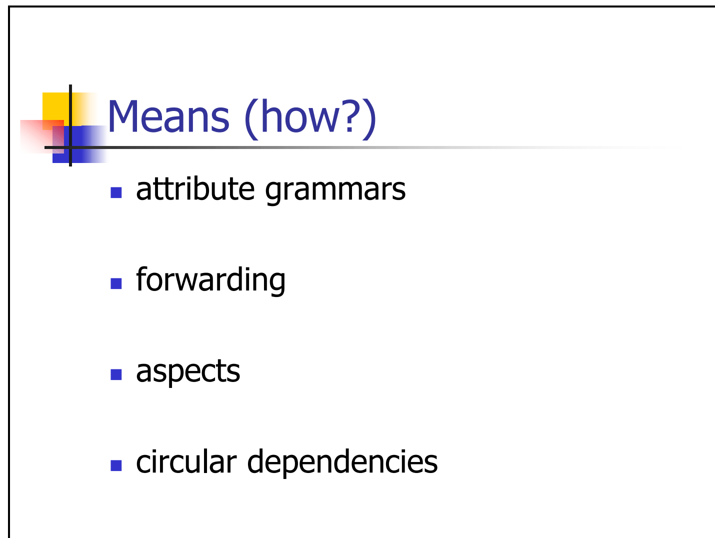
Let us now summarise the goals of intentional programming:

- Programming language features (such as array expressions) are independent components of the language implementation, that can be re-used freely in different combinations, to suit the programming task in hand.
- We wish to express optimisations that take us beyond the procedural and object-oriented abstraction mechanisms provided in present-day programming languages.
- Such optimisations often require knowledge of other language features, and therefore such knowledge should somehow be specified in the interface of an intention package.

These are ambitious goals, and indeed we are not alone in pursuing them. In particular, a recurrent theme in the literature is the use of macros for achieving extensible language implementations. Unfortunately macros often fall short on the third criterion, in the expression of the program analyses necessary to validate optimisations.

Let me slightly digress at this point, and address the main objection that has held back the introduction of programmer-supplied optimisations in source code. The objection has to be taken seriously, as it has been voiced by authorities such as Edsger Dijkstra and Don Knuth. Programmers need to have control over the efficiency of their programs. It is generally thought bad practice to rely on mechanised optimisation, especially if it could influence the asymptotic efficiency of your programs. The answer to this objection lies in those parts of intentional programming that I am not talking about today. Of course the programmer should be able to inspect and control the efficiency of her program, and not treat the optimiser as a black box. An interactive environment for intentional programming provides facilities for inspecting the translation of the source at various levels of abstraction.

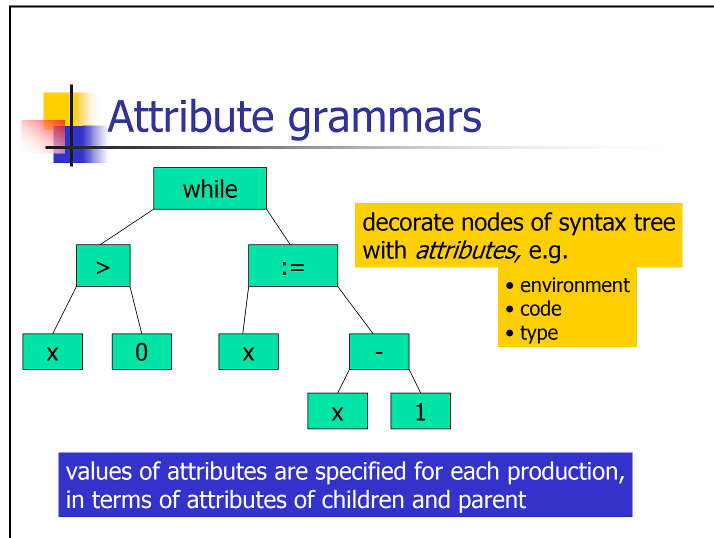
Slide 8



Back to the main topic of the talk. Now that we agree on *what* we want to do, I need to tell you *how* it is done. Before launching into the technical details, let me first give you a roadmap of our solution. Do not worry if the terminology sounds somewhat unfamiliar at this stage – I shall explain each of these concepts at leisure later on in the talk.

- Our starting point is a variety of *attribute grammars*. Attribute grammars were introduced in the late sixties by Don Knuth, as a way of expressing the semantics of programming languages with the same level of rigour as their syntax.
- While attribute grammars are certainly expressive, they are difficult to slice into components. When expressed as a pure attribute grammar, it would be necessary to know all other language features to introduce the new array feature. That would violate our requirement that the new feature is independent of other language extensions. Paul Kwiatkowski (of Simonyi's team at Microsoft) has solved this problem by introducing an elegant variation on inheritance, called *forwarding*.
- Another extension to attribute grammars is suggested by Kiczales' notion of *aspect-oriented programming*. I was glad to see that one of the previous speakers in this series has explained this exciting new direction in programming, so I can be brief. As in many other object-oriented systems, some components are best described as *aspects* that crosscut the traditional class hierarchy.
- Finally, we need to specify certain program analyses to verify the side conditions of optimising transformations. For instance, we shall need to define when an expression is free of side effects. When such analyses are written as an attribute grammar, they naturally lead to circular dependencies between attribute values. That may appear paradoxical, but as we shall see, such cycles have a well-defined semantics in the present context.

Slide 9



An attribute grammar is a description of the abstract syntax of a programming language, together with rules for decorating the abstract syntax tree with attributes. The attributes themselves are entities such as the environment (which variables are in scope?), the code (what assembly instructions does this tree compile to?), or the type of an expression.

The abstract syntax itself is described in the usual way, with a *production* for each separate feature of the language. For instance, the production for *while* states that *while* is an expression, and that it has two descendants (the condition and the loop body), both of which are expressions. We shall look at some further examples in a moment.

Because the nodes of the tree correspond to productions, the values of attributes are defined for each production separately. It is an essential property that the definitions can only access attributes of the immediate neighbours of a node in the tree, namely its parent and its children.

The attribute definitions do not specify in what order the values should be computed. One easy way that guarantees a correct order is to evaluate them on demand. Much of the literature on attribute grammars is concerned with more efficient evaluation schemes, but all of these are highly complex and it is not clear that they are much of an improvement over straightforward demand-driven evaluation. In this connection, it is significant that one of the few industrial attribute grammar systems (the Elegant system at Philips) is based on demand-driven evaluation. Furthermore, the performance of Elegant is on a par with other products that use more complex evaluation methods.The idea to use demand-driven evaluation for attribute grammars is due to Thomas Johnsson, now at Carlstedt Research and Technology AB in Gothenburg.

Slide 10

Example: mkarray

mkarray (size : expr, init : function) : expr

[type ← arrayType (init.returnType);
  pp ← "mkarray (" + size.pp + "," + init.pp + ")"
  ...
]

do we need to define *all* the usual attributes of expr?

Here is an example, namely the new production that describes our *mkarray* expressions. The upper box shows the abstract syntax: *mkarray* is an expression that takes two descendants. The first descendant is an expression itself and it is named *size*. The second child is a function (that maps indices to element values), and it is called *init*. In writing this production, we are deviating from the standard BNF notation because we wish to name individual descendants. The new notation also gives a nice analogy with function declarations, which is reflected in the demand-driven implementation.

We define two attributes here: the *type* of *mkarray*, and its textual representation named *pp* (short for *p*retty-*p*rinting). The type is defined to be an array type, and the elements of the array are of the type that is returned by the initialising function. The pretty-printing is defined somewhat naively by simple concatenation of strings.

We now face the question what the other attributes of *mkarray* should be: clearly, *mkarray* should define the same attributes as other expressions (because it can occur in any context where an expression is allowed). But what are the “usual attributes” of an expression? To maintain modular descriptions, we cannot define every attribute here once and for all, because we cannot know a priori what attributes may be introduced later, to support other language features.

Furthermore, it seems superfluous to give all these definitions of other attributes, because presumably they will already have been defined for the host language. Since *mkarray* is semantically equivalent to a program in the host language, could we not just borrow the definitions?

Slide 11

Forwarding

```
mkarray (size : expr, init : function) : expr      =
[ type ← arrayType (init.returnType);
  pp   ← "mkarray (" + size.pp + "," + init.pp + ")" ]
⇨ parse( {("$size",size),("$init",init)} ,
          "let     v := vec($size),
                   i := 0
           in      while (i < $size) do
                     v[i] := $init(i);
                     i := i+1
                   end;
                   return v
           end"    )
```

no optimisations

These considerations motivate the idea of *forwarding*, first formulated by Paul Kwiatkowski. Everything we need to write in order to implement *mkarray* (albeit without the optimisations) is shown on this slide.

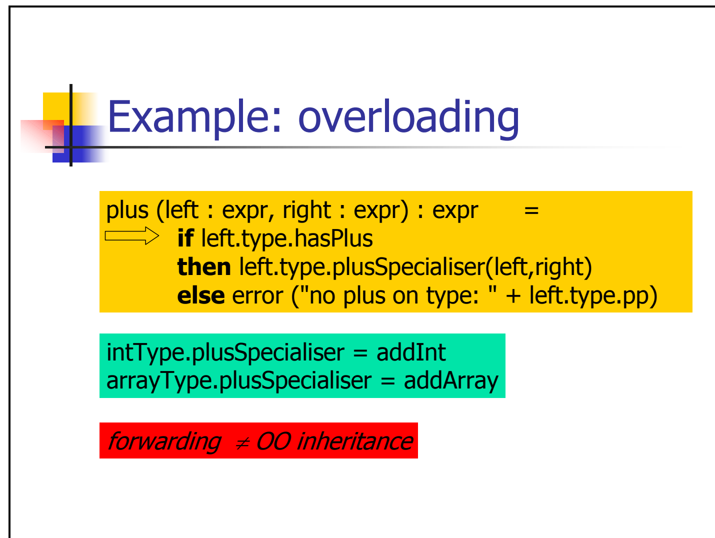
We define *type* and *pp*, but all queries for other attribute values are *forwarded* to another node in the syntax tree. Such forwarding is indicated by a fat arrow: the tree on the right-hand side of the arrow is the recipient of forwarded attribute queries. This recipient tree is constructed especially for the purpose, by parsing the equivalent piece of host program. The first argument to the parse routine specifies that the meta-variable *$size* should be bound to the first child of *mkarray*, and that *$init* should be bound to the other child. We thus create a tree that has the same parent as the original *mkarray*, using the descendants of *mkarray* in appropriate places lower down the new tree.

It is important that the effort of introducing *mkarray* in this way (as a language feature) is hardly greater than writing the equivalent procedure. What we have gained with a minimum of extra effort is the ability to do far more sophisticated processing of client code that uses *mkarray*, in particular by applying optimisations.

In this example, forwarding is very much like the traditional notion of inheritance: the production for *mkarray* could be viewed as a subclass of the production for let-bindings.

Some of you may be worried about variable naming: what if the names *v* and *i* already occur in the subprograms called *size* and *init*? The parser will generate fresh names instead, so *v* and *i* are merely placeholder identifiers.

Slide 12



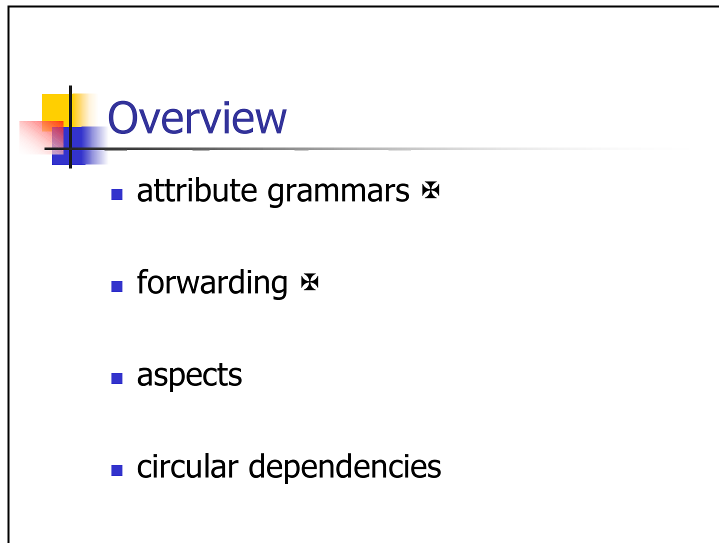
Let us examine another example of forwarding. Along with the *mkarray* construct, we naturally wish to introduce some basic vocabulary to manipulate arrays, including an addition operator that adds two arrays of the same size by adding corresponding elements. That addition operator is represented by the (+) operator, and therefore we wish to overload the (+) symbol.

To achieve this, every type that is "addable" has a special attribute named *hasPlus*, which is set to *true*. For all other types, the value of *hasPlus* is *false*. (Of course that default value is defined only once for all non-addable types, but because of time constraints I shall omit details.) Whenever *hasPlus=true*, there exists another attribute called *plusSpecialiser*. This attribute is a function, that when given two children, builds a new tree to which all attribute queries are forwarded.

The *plusSpecialiser* of the integer type is addition of integers; for arrays, it is addition of arrays. What is nice about this implementation of overloading is its highly local nature. To overload (+) on another type (for instance strings, where addition is concatenation) you only need to change the relevant type, and nothing else.

The above example demonstrates how forwarding is different from ordinary inheritance, because the specialisation can be achieved only dynamically, when we have a tree to manipulate. There have been other attempts to use object-orientation for structuring attribute grammars, notably by Aksit and Mernik. These attempts allow only static inheritance. The idea of forwarding is much closer to the *higher-order attribute grammars* of Teitelbaum, Swierstra and others. In a higher-order attribute grammar, attributes may be attributed trees themselves, and this is precisely what happens in our implementation of forwarding. Higher-order attribute grammars are an important feature of the Synthesiser Generator, a product of GrammaTech. Indeed, if you want to separate the wheat from the chaff in attribute grammars, restricting your attention to industrial products is a good start.

Slide 13



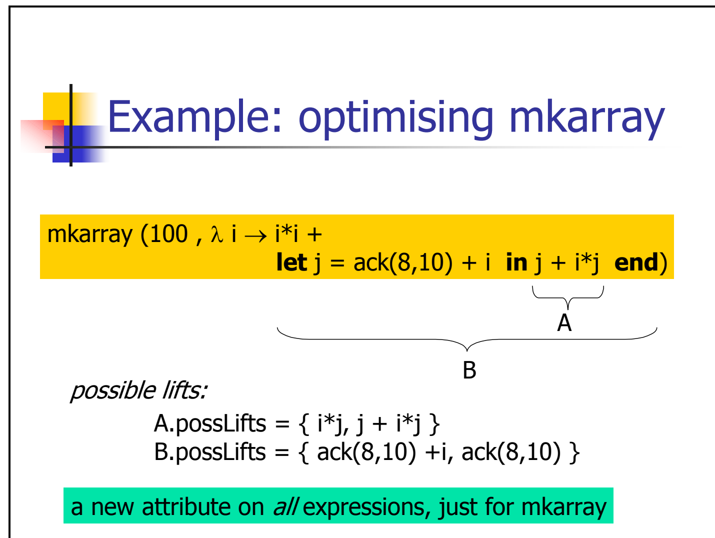
It is time to pause for a bit, and take stock of what has been achieved so far.

- We have very briefly introduced *attribute grammars*. If you have been baffled by previous expositions of the subject, the message is not to worry. We simply evaluate the attribute definitions through demand-driven evaluation. Also, the traditional distinction between inherited and synthesised attributes is unimportant here.
- The main innovation of intentional programming over traditional attribute grammars is the notion of *forwarding*. Forwarding is used in the same way as inheritance, but also to choose at compile-time between different implementations of the same piece of syntax. In particular, forwarding can be used to implement overloading, and also to implement optimising transformations.

It is interesting to speculate why forwarding was not invented earlier. I believe this is in large part due to the emphasis in the attribute grammar community on efficient evaluation mechanisms and very thorough static checking. By comparison, little effort has gone into better ways of structuring attribute grammar descriptions. The efforts of Aksit and Mernik I mentioned before are rare exceptions, as is the work of Waite and Kastens. Like the other big innovation in attribute grammars (Johnsson's demand-driven evaluation), the idea of forwarding came from an outsider, one of the chief designers of intentional programming, Paul Kwiatkowski.

To some extent I have been short-changing you, because I have not shown at all how the optimisation of array expressions is carried out. And yet I claimed that this was the main reason for introducing arrays as an intention, and not as a procedure! At this point I have to admit that the descriptions of the optimisations are a bit too complex to be shown in their entirety here, but I will try to highlight the main features of intentional programming that enable us to code them in an effective manner: *aspects* and *circular dependencies.*

Slide 14



To optimise *mkarray*, all other expressions will need to have an attribute that is a set of subexpressions. A subexpression is in this set precisely when it is a candidate for hoisting by an enclosing *mkarray*. We call this attribute the *possible lifts* of an expression.

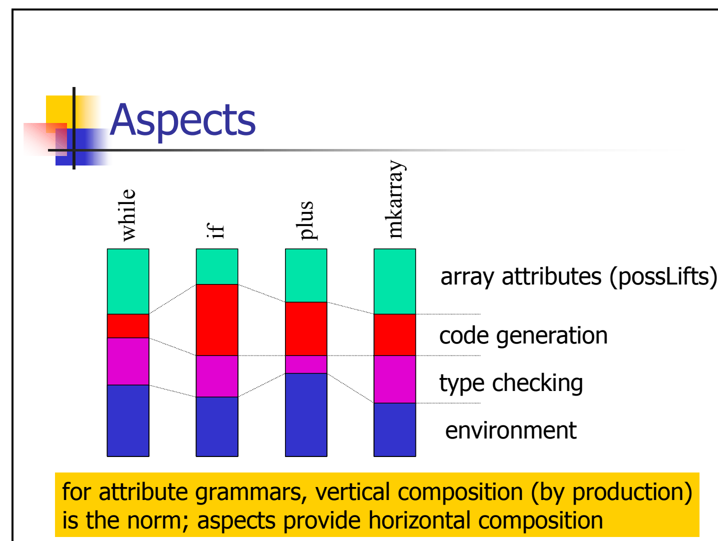
A possible lift *P* of a *mkarray* expression *E* (that is $P \in E.possLifts$) is abstracted into a let-binding if there is no enclosing *mkarray* that also abstracts *P*.

To illustrate, consider the expression at the top of this slide. I have labelled two sample subexpressions *A* and *B*. Expression *A* has two possible lifts, namely *i*j* and *j+i*j*. Neither of these is a possible lift of *B*, however, because hoisting either out of *B* would take the variable *j* out of scope. The only possible lifts of *B* are the constant expression *ack(8,10+i)* and *ack(8,10)*. Indeed, the latter expression will be hoisted out of the enclosing *mkarray*, because it is independent of the index variable *i*.

As I have mentioned before, it is not acceptable to change the code of all other expression productions merely because we decided to optimise *mkarray* expressions. Instead, we group the definitions of *possLifts* together in a so-called *aspect* that can be *woven* with the existing productions.

Such an aspect defines *possLifts* separately for each production in the host language. Provided all new forms of expression eventually forward to primitives in the host language, this makes the description of *mkarray* an independent component. Let me say this once again, because it is important: to ensure that *possLifts* is always defined (even for new intentions that are introduced later) we require two things: first, all elements of the host language are given definitions for it. Second, all intentions forward (eventually) to trees composed only of host language intentions.

Slide 15



Here is a pictorial explanation of aspects, where columns correspond to productions. Traditionally, attribute grammars are organised by production. Each production provides definitions of attributes for the environment, for type checking, for code generation – and in our example, for optimising array expressions. You could think of productions as class declarations. To add another attribute, we derive a new class for each production, thus extending each column vertically.

In an aspect-oriented organisation, we can also compose an attribute grammar horizontally, by providing a slice (that cuts across the production boundaries). Such a slice is called an *aspect*, and it provides definitions for a set of related attributes. In the picture, I have indicated different aspects by different colours. An *aspect weaver* takes the individual aspects, collecting the bits and pieces that contribute to individual productions, to produce a traditional attribute grammar.

The description of the host language is typically organised in the traditional way, by grouping attribute definitions in productions. The addition of later features, such as the optimising *mkarray*, is described by means of aspects. Within each aspect, one gives definitions only for the relevant attributes. Any attributes that are used but not defined are recorded in its interface, so that it is possible to check statically whether a set of aspects, when woven, will produce a well-defined attribute grammar.

To some members of the audience, it may be helpful to note that this particular form of aspects is reminiscent of the *mix in layers* found in C++.

Slide 16

Example: purity test

aspect purity : *attribute declaration*

pure : max(bool);

letbind (lhs : id, rhs : expr, body : expr) : expr =
[pure = rhs.pure ∧ body.pure]

equation, not definition

add (left : expr, right : expr) : expr =
[pure = left.pure ∧ right.pure]

varref (var : id, dcl : expr) : expr =
[pure = dcl.rhs.pure]

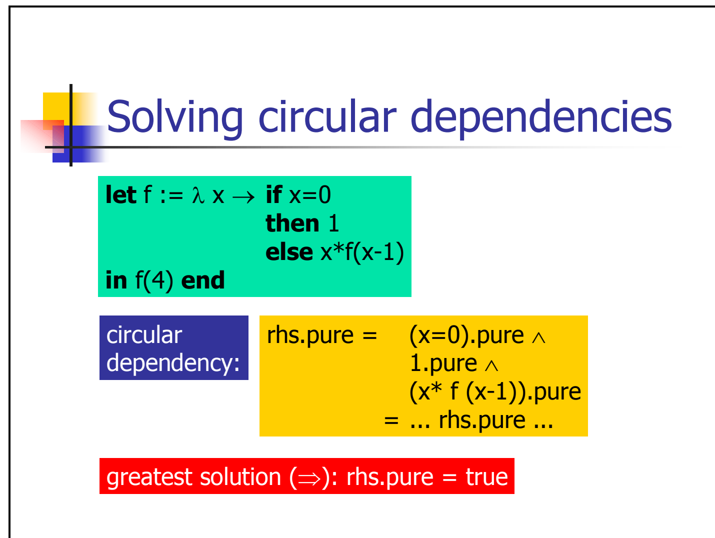
Apart from the *possible lifts*, another aspect that needs defining is a test for the absence of side effects. As I mentioned earlier, the optimisations of *mkarray* are valid only in the absence of side effects such as *print* statements and assignments.

We therefore introduce a new attribute *pure* that indicates whether an expression is free of side effects or not. The relevant aspect starts by declaring *pure* to be a Boolean value; it also expresses the fact that we are interested in the maximum Boolean value (in the implication order) that satisfies the equational constraints stated below:

- The first constraint says that a let-binding is pure if its right-hand side is pure, and its body is pure.
- The second constraint says that an addition is pure if both arguments are pure.
- Finally, a reference to a let-bound variable is pure if the right-hand side of the let-binding is pure. Here we use the fact that the declaration of a reference always appears as a child below the reference. This idea (which is easily implemented by forwarding) is shared with the *reference attribute grammars* of Görel Hedin.

You will have noticed that I have carefully avoided calling these constraints *definitions*, and I used an equality sign rather than an assignment symbol. And what is this business of looking for a *maximum* solution to the equations, rather than the *unique* solution?

Slide 17



Consider this simple example, where *f* is a recursively-defined function.

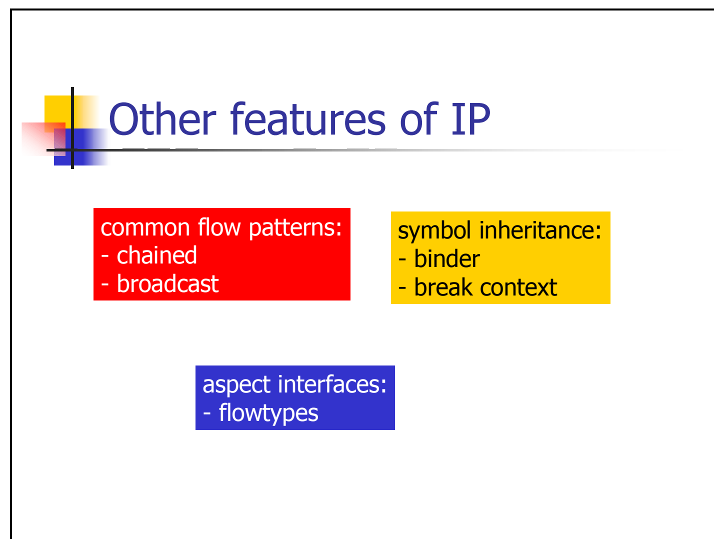
Is this a pure expression? The equations I gave before say that the right-hand side of *f*'s definition is pure if all its components are pure – but those components include a reference to *f* itself! And that reference is pure if the right-hand side is pure, so we conclude that there is a cycle in this chain of equations.

The right way to view this phenomenon is that we have a set of equations that are constraints on instances of the *pure* attribute. That set of equations potentially has more than one solution: among the different alternatives, we are interested in the weakest solution. In this example, that means we have defined *rhs.pure=true*.

This type of circular dependency is resolved by first computing all attributes that are used in the right-hand sides of the relevant equations, and then solving the constraints separately using *fixpoint iteration*. Fixpoint iteration is a general method for finding the greatest (or smallest) solution of a set of equations.

When Knuth invented attribute grammars, he was at pains to exclude these circular dependencies, perhaps because at that time the theory behind fix points in computer science was still the domain of a few specialists. The observation that circular attribute equations are useful for program analysis is due to Rodney Farrow. Again it is a good idea that has proven its worth in an industrial context, this time in products called *JavaAuditor* and *C++Auditor* that are marketed by *Declarative Systems Inc*. Our own interest was stimulated by discussions with colleagues at Microsoft on an experimental feature of intentional programming called *rollback*, which seeks to address similar problems.

Slide 18



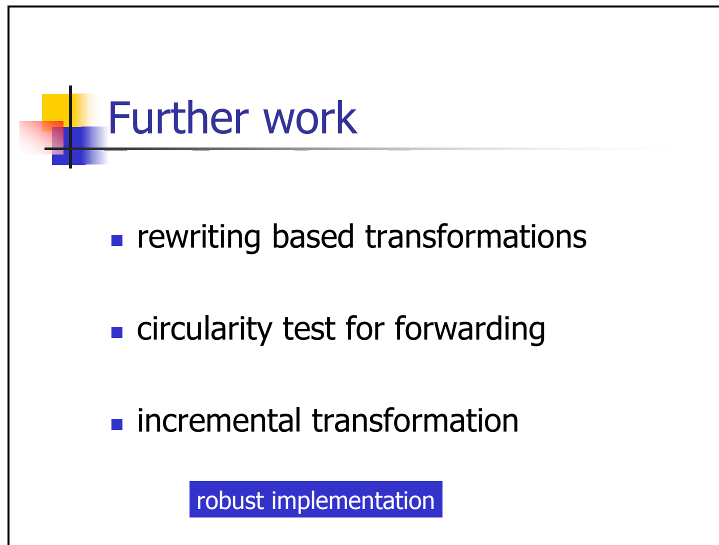
This concludes our whirlwind tour of the main concepts that underlie intentional programming. Because of time constraints, I have not been able to tell you about certain other ideas, so let me just give you a couple of brief headlines.

Often an attribute flows in a particular way through the tree: for instance, a common pattern is that of *chained attributes*, which are passed from left to right through the children of each node. Another common pattern is that of *broadcasting* a particular value down a subtree. Each of these patterns can be defined as a template, a generic definition of attributes.

Similarly, many symbols share the same behaviour. For instance, all variable binders are treated in a similar way, and all contexts for a *break* statement share much of their code. Again this can be coded through templates. These features are described in some more detail in my invited paper at last year's International Workshop on Attribute Grammars and their Applications.

Perhaps the most important feature that I have skimmed over is the *interface* of an aspect. Such an interface states for each type of language element (such as expressions or statements) what attributes are defined, and what attributes are assumed to be defined elsewhere. This is called the *flowtype* of the language element. There are some subtle interactions with type inference here, and as yet the design is not fully complete.

Slide 19



I would like to conclude by mentioning a number of areas where intentional programming still needs to be improved further:

- In the present design it can be a little awkward to encode optimisations that are based on rewriting. As we have seen, it is fairly easy to specify transformations that are essentially a form of conditional macro expansion. It is not so easy, however, to specify pattern matching in this style. Ganesh Sittampalam and Ivan Sanabria-Piretti have been studying particular applications of such rewriting transformations, and the integration of their results with the current design of intentional programming is the subject of ongoing research.

- It is rather easy to make mistakes with forwarding, and inadvertently create cycles in the attribute dependencies. Such mistakes are very difficult to track down. We hope that existing algorithms that check for cycles in higher-order attribute grammars can be adapted to forwarding. Kevin Backhouse is currently working out the details.

- To apply these techniques to large systems, it is important that the transformation process can be executed incrementally. This is again a little more difficult than for more traditional forms of attribute grammars. Ganesh Sittampalam, Jens Peter Secher and myself have recently started work on this problem, but it is too early to report results.

We need a robust reference implementation that is suitable for experimentation by others. Eric van Wyk has built a number of small prototypes to explore the initial ideas. Now the time is ripe to tie everything together, and build a system that can be shared with others, in particular with our colleagues at Microsoft Redmond, who have already built an impressive interactive environment for intentional programming.