Programming Research Group

ABSTRACTS FOR THE PRG STUDENT CONFERENCE 2000

Edited by Ranko Lazic and Luke Ong

PRG-TR-15-00

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

The 2000 vintage PRG student conference, like much low-yield wine from established regions, is of good quality with the prospect of lasting well. Represented in the harvest is the new variety of Programming Tools and the old cultivar of Functional Programming. Several other varieties unfortunately did not ripen in time this year.

We thank Richard Bird for his aperitif opening of the conference; Samson Abramsky for a sparkling invited talk: *Interaction: splitting the atom of computation*; Wendy Adams for fortified assistance with making the day a success and for compilation of tex files; and the recent CLaSS scheme for funding the prizes for best abstract and best talk.

The conference continues to complement weekly cakes talks. Some presentations were honed versions of cakes talks; others were chosen by their authors to offer a change of topic. They are all welcome.

To the future of our graduate students!

Ranko Lazic, Luke Ong and Jeff Sanders
Organising Committee

Abstracts

Kevin Backhouse	*Fixing program analysis problems* (supervisor: B. A. Sufrin)
Ian Bayley	*Nested versus regular datatypes* (supervisor: R. S. Bird)
Christie Bolton	*On giving a behavioural semantics to activity graphs* (supervisor: J. Davies)
Yorck Hünke	*Programming with dependent types* (supervisor: O. de Moor)
Jan Jürjens	*Bridging the gap: ... reasoning about cryptography* (supervisor: S. Abramsky)
David Lacey	*Imperative program transformation by rewriting* (supervisor: J. M. Spivey)
Shin-Cheng Mu	*On inverting functions* (supervisor: R. S. Bird)
Andrzej Murawski	*Machine-independent characterizations of complexity classes* (supervisor: L. Ong)
Iván Sanabria-Piretti	*Transforming abstract data types by rewriting* (supervisor: J. M. Spivey)
Silvija Seres	*Re-entrant locks in Java* (supervisors: C. A. R. Hoare & J. M. Spivey)
Ganesh Sittampalam	*Annotating programs for efficiency* (supervisor: O. de Moor)
Paolo Zuliani	*Reasoning about quantum-mechanical nonlocality* (supervisor: J. W. Sanders)

Fixing Program Analysis Problems

Kevin Backhouse

Program analysis is an important technique in compiler design. By analysing a computer program, the compiler can often deduce a number of useful properties. For example, analysis is used for "dead code elimination" and "constant propagation". Results from the analysis can be used to optimise the program or warn the programmer about possible errors. Due to their use in compilers, it is important that program analysis algorithms should be guaranteed to terminate. As a result, the analyses often produce conservative rather than exact results and the properties being computed are not terribly sophisticated. Despite this, the results can be very useful.

The basis of many program analysis algorithms is the computation of a *least fixed point*. I will demonstrate this with an example: analysing the exception usage in an imperative program. In my talk, I will discuss ways of specifying such algorithms and computing them efficiently.

An example: analysing exception usage

Suppose we have an imperative programming language in which we can throw and catch exceptions. We would like to infer the set of exceptions that might be thrown by procedures in the program. This will allow us to alert the programmer to potentially uncaught exceptions. Here is a sample program with two procedures:

procedure f {	**procedure** g {
raise e_1	**raise** e_3
try {	**call** f
call g	**raise** e_2
} **with** $e_3 \to$ **call** f	}
}	

By manually examining these procedures, we can see that the sets of exceptions that might be raised by f and g are $\{e_1, e_2\}$ and $\{e_1, e_2, e_3\}$ respectively. So, if f is the entry point of the program, then we would warn the programmer that e_1 and e_2 might not be caught.

How would we write a computer program to infer these sets of exceptions? The standard approach in program analysis [1] is to reduce the program to a set of equations. The variables in the equations are X_f and X_g, which are the sets of exceptions that might be raised by f and g, respectively:

$$\begin{aligned} X_f &= \{e_1\} \cup (X_g \backslash \{e_3\}) \cup X_f \\ X_g &= \{e_3\} \cup X_f \cup \{e_2\}. \end{aligned}$$

Next, we compute the least solution to these equations, which is: $X_f = \{e_1, e_2\} \ \wedge \ X_g = \{e_1, e_2, e_3\}$. The fact that we choose the *least* solution is important: otherwise we might include unnecessary elements in X_f and X_g. For example, $X_f = \{e_1, e_2, e_3\} \ \wedge \ X_g = \{e_1, e_2, e_3\}$ is also a valid solution to the equations.

Least Fixed Points

In the example given above, we saw that the solution to the analysis problem was the least solution of a set of simultaneous equations. Most programming languages have a mechanism for computing least fixed points. For example, the language Haskell allows us to write the equations for X_f and X_g almost exactly as we stated them above:

```
x1 = ["e1"] `union` (x2 \\ ["e3"]) `union` ["e2"]
x2 = ["e3"] `union` x1 `union` ["e1"]
```

Unfortunately, this Haskell program is wrong. Rather than computing the two sets that we expected, it goes into an infinite loop. As I will explain in my talk, this is because the semantics of Haskell's least fixed point operator is not the semantics that we want. In the context of Attribute Grammars, Farrow [2] has suggested that we could obtain correct results by simply using a different fixpoint operator. In my talk I will argue that this idea is equally applicable in the world of functional programming.

References

[1] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. ACM Computing Surveys, 18, 3, 277–316, sep, 1986.

[2] R. Farrow. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. SIGPlan '86 Symposium on Compiler Construction, Association for Computing Machinery, SIGPlan, Palo Alto, CA, jun, 1986, 85–98.

Nested versus Regular Datatypes

Ian Bayley

A parameterised type is *regular* if its parameters do not change from the left to the right-hand side of its Haskell definition. *Nested* types generalise regular types and are allowed to have different parameters on either side. Here are two nested datatypes, of which *Tree* is regular and *Nest* is not:

$$\begin{array}{l}\textbf{data}\, \mathit{Tree}\ a = \mathit{Null} \mid \mathit{Branch}\ (\mathit{Tree}\ a, a, \mathit{Tree}\ a) \\ \textbf{data}\, \mathit{Nest}\ a = \mathit{NilN} \mid \mathit{ConsN}\ (a, \mathit{Nest}\ (a, a))\end{array}$$

Each value of $\mathit{Nest}\ a$ is a peculiar "list" whose first element has type a, second element has type (a, a), third element has type $((a, a), (a, a))$ and so on. We can think of each value as representing an internally-labelled, perfect binary tree. (A tree is perfect if the left and right subtrees of each node have the same height). Each successive element represents the next row of the tree. So, in this sense, $\mathit{Nest}\ a$ represents a subtype of $\mathit{Tree}\ a$, the type of internally-labelled binary trees of as. An example instance of $\mathit{Nest}\ \mathit{Int}$ is

$$\mathit{ConsN}\ (1, \mathit{ConsN}\ ((2,3), \mathit{ConsN}\ (((4,5),(6,7)),\ \mathit{NilN})))\ .$$

Suppose we attach to a Haskell function f the type signature $f :: X \rightarrow \mathit{Tree}\ a$. This is a partial specification of f, so we expect the function always to return a *Tree*. If it does not then the type checker will tell us so, hinting that we have implemented f wrongly. Errors that would otherwise have been detected only at run-time (if at all) are now spotted earlier at compile-time instead.

This use of type signatures is standard but we can take it further. We can trap even more programmer errors at compile-time by using more restrictive (and therefore more informative) types. This is because nested datatypes capture constraints on the use of regular datatypes. Suppose we want f to return a strictly perfect *Tree*. We can ensure this by trying to write a new f with type signature $f : X \rightarrow \mathit{Nest}\ a$; we will thereby trap at compile-time any function that can return imperfect trees.

Another subtype of *Tree* that we can capture is that of AVL trees: trees which are almost balanced in that the left and right subtrees of each node differ in height by at most one.

$$\begin{array}{lcl}\textbf{type}\, \mathit{AVL}\ a & = & \mathit{AVL}'\ a\ ()\ a \\ \\ \textbf{data}\, \mathit{AVL}'\ a\ l\ h & = & \mathit{AVLZero}\ l \\ & \mid & \mathit{AVLSucc}\ (\mathit{AVL}'\ a\ h\ (\mathit{AVLNode}\ a\ l\ h)) \\ \\ \textbf{data}\, \mathit{AVLNode}\ a\ l\ h & = & \mathit{LeftLarger}\ h\ a\ l \\ & \mid & \mathit{EqualHeight}\ h\ a\ h \\ & \mid & \mathit{RightLarger}\ l\ a\ h\end{array}$$

This datatype and the next one were devised by Ross Paterson but left unpublished. An example instance is

$$AVLSucc\ (AVLSucc\ (AVLZero\ (LeftLarger\ 1\,2\,()))) \ .$$

By removing the *LeftLarger* and *RightLarger* cases, we can get another datatype of perfect trees. However, this one yields elements in a manner suited to depth-first traversal, whereas *Nest* reveals them in a breadth-first manner. We can also capture as a nested datatype, the constraint on lists of lists that they must form a matrix.

$$\begin{array}{lcl}
\mathbf{type}\ Matrix\ a & = & VecOfVec\ a\ a \\
\\
\mathbf{data}\ VecOfVec\ a\ v & = & One\ (Vec\ v\ v) \\
 & \mid & Cons\ (VecOfVec\ a\ \ (a, v)) \\
\\
\mathbf{data}\ Vec\ a\ v & = & One'\ v \\
 & \mid & Cons'\ (Vec\ a\ \ (a, v))
\end{array}$$

Observe that $Vec\ a\ a$ produces a vector of elements of a so $VecOfVec\ a\ a$ produces a vector of vs where the vs are themselves vectors. Here is a 2×3 matrix of type *Matrix Int*:

$$(Cons \cdot Cons \cdot One \cdot Cons' \cdot One')\ ((1, (2, 3)), (4, (5, 6)))$$

Note that the dimensions are encoded by the constructors.

In the talk, we will see how to use type constructor parameters to form strictly square matrices. Programming with nested datatypes is often tricky, because a recursive function defined on *Nest a* for example, calls itself with input $Nest\ \ (Pair\ a)$. We call this phenomenon *polymorphic recursion*. Finally we will see how folds and other generic combinators can be used to make the task of programming simpler.

On giving a behavioural semantics to activity graphs

Christie Bolton

The Unified Modeling Language (UML) is a collection of graphical notations which was introduced to be the de facto industry standard language for the specification and development of object-oriented systems. It can be used to describe both the structure and the behaviour of a system at various levels of rigour or abstraction. Whilst UML has a precisely-defined syntax and static semantics its behavioural semantics is specified only informally.

In this talk we explore how formal description techniques may be used to reason about UML specifications; how descriptions of system behaviour, expressed in terms of abstract data types and processes, may be constructed from individual UML diagrams. Given these formal interpretations, and the associated behavioural and relational semantics, we show how existing refinement techniques may be applied to support the further development and analysis of object models.

Our intention is not, however, to give a complete or formal semantics to the whole of UML. Indeed, perhaps that is not an appropriate thing to do. Before we embark on the formalisation process it is important to consider the role of the various notations within the language.

The use of UML in software specification and design can be divided into three, not necessarily distinct, phases: an initial brainstorming period, the development process, and the implementation-level specification. Diagrams constructed in the first phase—for instance CRC cards and use case diagrams—are a means of getting ideas flowing and opening up a dialogue between various members of the team. A formal semantics is less useful here.

We focus on diagrams constructed in the last two phases. Activity diagrams and sequence diagrams constructed during the development process describe the behaviour of the system. A combination of class diagrams and state diagrams comprise the final class description, or implementation-level specification.

We begin this talk by presenting a formal semantics for activity graphs, describing them in terms of the process language CSP. We go on to explain how this semantic interpretation may be used to verify that the implementation of a system is consistent with its specification. We discuss how this verification might be automated.

Activity graphs

Activity graphs are used to describe the flow of behaviour within a system. They combine features of Harel State Charts with the intuition of Petri Nets. An activity graph is constructed from a combination of action states, (sub)activity states, start and finish states and pseudostates merge, decision, fork and join.

The process corresponding to each *state* within an activity graph is essentially the sequential composition of the process corresponding to its incoming lines, the process corresponding to its type and the process corresponding to its outgoing transitions. The resulting process is then interleaved with itself to facilitate multiple (possibly synchronous) executions

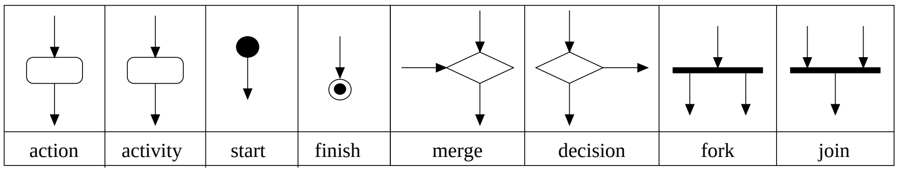

Figure 1: The states of an activity graph

of the same state. More formally this process, $StateP(state)$ is given by[1]:

$$LinesInP(state)(\; TypeP(state)\; TransOutP(state)$$

$$StateP(state)\,)$$

The definitions of the processes corresponding to the incoming lines, outgoing transitions and type of each state, $LinesInP$, $TransOutP$ and $TypeP$ respectively, depend on the type of the state. For instance *join* states require all in-coming lines to be enabled whereas all other states (with the exception of the start state which has no incoming lines) require precisely one of their incoming lines to be enabled.

The process corresponding to a *graph* is then the parallel composition of the processes corresponding to each of its constituent states, each synchronising on its own alphabet or set of associated events.

$$GraphP \quad = \quad \| \;\; state : States \;\; [\, alphabet(state)\,] \;\; StateP(state)$$

Our semantic function takes an activity label pointing to a particular graph and returns the process which models the *behaviour* of the given graph. This is the process obtained by hiding all the line events in the process $GraphP$ corresponding to the given graph.

Analysis and future work

One of the strengths of UML, why we hope it will lead to a general improvement in the quality of program design, is that its graphical nature is appealing to software developers with no background in formal methods, those who might otherwise skip the specification process and move directly to the coding phase. It is therefore crucial in giving a formal semantics to various parts of UML that we consider the needs of these users. To expect them to learn the more traditional formal methods in order to verify the consistency of their UML specifications is unrealistic.

In order to compare the final class description of a model with the activity graphs constructed during the development process we must first translate the final class description,

[1] For brevity and clarity, the definition given is a simplification. We take care to handle self-transitions separately. Furthermore, to prevent the process from diverging, we put it in parallel with a constraining process which bounds the number of synchronous executions of any given state.

expressed as a combination of class diagrams and state diagrams, to its corresponding process. Having translated both the final class description and the behavioural descriptions constructed during the development process to the same language it is simply a question of refinement to verify that our model is consistent.

We conclude this talk by identifying such a translation from the final class description of a system, via an equivalent abstract data type, to its corresponding process. We raise some of the issues that still need to be addressed before this step is fully automatable. Our long term goal is for a modelling tool to translate automatically the graphical descriptions of the final class model of a system and specifications of its desired behaviour to their corresponding processes, to verify consistency using a model checker, and to give the user intelligible reports—in terms of the methods on the classes—about where/if the specification is violated.

Programming with dependent types

Yorck Hünke

Type systems have become more sophisticated in recent years. For example, Haskell extends Hindley-Milner type checking with type classes in order to handle ad-hoc polymorphism. In this talk, we consider another extension: dependent types — this extension allows types to depend on values. We take a look at programs which cannot easily be written in Haskell because of type limitations and present solutions using the new functional language Cayenne [1] which provides type checking for dependent types. Further example programs can be found in [3], upon which this talk is based.

Limitations of Haskell

Despite Haskell's extensions to conventional type systems, programmers still do not have the flexibility they require. Consider the function `printf`, familiar from the C programming language.

```
? printf "%i is an integer, %s is a string." 3 "ABC"
> 3 is an integer, ABC is a string.
```

The function `printf` takes a string as its first argument, which contains literal and formatting characters. The number of further arguments and their types depend on the *value* of the first argument; such a dependency between types and values cannot be expressed in Haskell.

A similar example is the function `zipwith`, defined informally as follows.

```
zipwith :: (t1 -> ... -> tn -> t) -> [t1] -> ... -> [tn] -> [t]
zipwith f (a1:as1) ... (an:asn) = f a1 ... an : zipwith f as1 ... asn
zipwith f  _        ...  _       = []
```

Although it is possible to define `zipwith` for any fixed `n`, such an implementation is somewhat clumsy because a separate definition of `zipwith` is needed for each `n`. However, it is not possible to define a parametrized `zipwith` in Haskell in the form above because the type of `zipwith` depends on the value of the parameter `n`.

Dependent types

Dependent types allow the programmer to express in the type system what seems natural in the examples given, namely that types can depend on values.

We evaluate dependent types as an extension to conventional type systems using the functional programming language Cayenne [1] which provides type checking for dependent types. Dependent types show up in two of the type rules in Cayenne: in function application, the type of `f a` depends on the value of `a`; in the rule for the `case x of` construct, the types of the branches depend on the value of `x`. Details of Cayenne's type system can be found in [1].

Types in Cayenne are treated as first-class objects, hence the type of types `#` is a visible part of the language, and type annotations are compulsory. Further differences between Cayenne and Haskell are best explained using an example.

```
map :: (a :: #) -> (b :: #) -> (a -> b) -> List a -> List b
map a b f (Nil)  = Nil
map a b f (x:xs) = f x : map a b f xs
```

Type arguments have to be made explicit in all polymorphic type definitions in Cayenne. Furthermore, the value of an argument can be given a name in the type definition, and referred to on the right of it. Here, the first two arguments `a` and `b` of the function `map` are of type `#` and represent the input and output types of the function `f`, the third argument of `map`.

Case studies

Our evaluation of dependent types consists of several case studies. First, we define a function `neg` which performs negation but can operate on values of different types, something similar to Haskell's overloading mechanism. Then, we consider a function `sum` which sums the integer components of a data type. This function can be defined for a range of data types in Haskell, e.g. pairs and trees, but for each of these data types, a separate definition of `sum` is needed. Dependent types can be used to write a single definition of `sum` which can operate over a range of data types. A similar technique can be used to implement the function `printf`.

Our main case study consists of a parametrized form of `zipwith`. Although the definition of a generalised `zipwith` in [2] illustrates that it is possible to implement `zipwith` with a valid type signature in a polymorphic type system without dependent types (by using continuations), dependent types allow us to assign a more precise type to `zipwith` than that described in [2] by operating over a data type of lists indexed by their length.

In summary, dependent types combine two concepts: they provide more flexibility, i.e. more programs can be expressed in the language and pass the type checker, but at the same time they allow the programmer to capture more precise information about the functionality of programs in their type.

References

[1] Lennart Augustsson. *Cayenne - a language with dependent types.* Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), SIGPLAN Notices, 34(1):239-250, January 1999.

[2] Daniel Fridlender, Mia Indrika. *Functional Pearl: Do we need dependent types?* Journal of Functional Programming, 10(4):409-415, July 2000.

[3] Yorck Hünke. *Programming with Dependent Types.* Undergraduate dissertation, supervised by Oege de Moor, Oxford University Computing Laboratory, Oxford University, May 2000.

Bridging the Gap: Formal vs. Complexity-Theoretical Reasoning about Cryptography

Martín Abadi
Bell Labs Research
Lucent Technologies
&
Jan Jürjens
Computing Laboratory
University of Oxford

The work reported in this talk is part of an effort to bridge the gap between the formal and the complexity-theoretical views of cryptography and cryptographic protocols. The formal view is based on ideas and methods from formal methods (e.g., Dolev, Yao 1983; Burrows, Abadi, Needham 1989; Lowe 1995; Roscoe 1997; Paulson 1998)[2]. It has led, in particular, to techniques for high-level reasoning (even automated reasoning) about security protocols. The complexity-theoretical view has developed sophisticated foundations and proof techniques (e.g., Blum, Micali 1982; Yao 1982; Goldwasser, Micali 1984). Several recent projects have made progress in the direction of linking these two views (Abadi, Rogaway 2000; Pfitzmann et al. 2000; Lincoln et al. 1998). These projects differ in their approaches and their technical details. Overall, however, they aim to refine and to provide foundations for formal treatments of cryptography, and to combine the benefits of both treatments for reasoning about systems that use cryptography.

More specifically, this work extends the recent work of Abadi and Rogaway, which compares a formal account of symmetric encryption with a complexity-theoretical account. Both of these accounts are fairly standard in the respective communities. That work deals with expressions that represent pieces of data: booleans, pairs, randomly generated keys, and symmetric encryptions. Each expression induces an ensemble (that is, for each choice of a security parameter, a probability distribution over the bitstrings that the expression represents). The main result there is that if two expressions are equivalent in the formal model then the ensembles that they induce are complexity-theoretically indistinguishable. The result means that formal reasoning about equivalence of expressions is sound with respect to the complexity-theoretical model. Furthermore, the hypotheses of the result indicate a few significant discrepancies between the two accounts (for example, in the treatment of encryption cycles).

Here we go further by considering systems that use symmetric encryption and decryption. The systems may in particular represent cryptographic protocols with several parties. We describe systems in a simple specification language representative of those commonly used in formal reasoning about security. We present the two models as two semantics for

[2] For these and the following references and for basic definitions cf. M. Abadi, P. Rogaway: "Reconciling Two Views of Cryptography", IFIP-TCS 2000, LNCS 1872, pp. 3–22, http://pa.bell-labs.com/~abadi/allpapers.html#csds

the language. In both the formal model and the complexity-theoretical model, systems generate traces. Traces are sequences of tuples of values (one value for each communication channel at each time instant), and two systems are equivalent if an eavesdropping adversary cannot distinguish a trace generated by one from a trace generated by the other. In the formal model, a value is a formal expression; the adversary is allowed only simple symbolic computations over expressions. In the complexity-theoretical model, a value is a bitstring; the adversary is a resource-bounded Turing machine that computes over bitstrings.

We obtain a soundness result analogous to that of Abadi and Rogaway: if two systems are equivalent in the formal model then the probability ensembles that they induce are complexity-theoretically indistinguishable. This result means that formal reasoning about equivalence of systems is sound with respect to the complexity-theoretical model, and that complexity-theoretical eavesdroppers do not have more power than the simpler formal eavesdroppers.

This study of eavesdroppers presents difficulties and offers some insights, beyond those encountered in previous work. In particular, we treat situations where exactly the same ciphertext appears twice in a piece of data, and the adversary can notice a repetition. In addition, we analyze programs where the control flow depends on the properties of the underlying cryptosystem, for example a program that generates two keys, compares them, and branches on the result of the comparison. In such examples, the formal model and the complexity-theoretical model do not give rise to exactly corresponding traces, but we show that the differences are negligible (in a precise sense). Finally, we consider a substantial difference between the formal and the complexity-theoretical accounts of decryption. Formal models that treat symmetric encryption commonly assume that decrypting a message with a key K succeeds only if the message was encrypted under K, and that the success or failure of a decryption is evident to whoever performs it. This property is convenient in both protocol design and analysis; one may even argue that “encryption without integrity checking is all but useless” (Bellovin 1996). The property can be implemented by means of checkable redundancy. Complexity-theoretical models typically do not build it in. One outcome of our work is a precise complexity-theoretical counterpart of this property.

While the present work focuses on eavesdroppers and symmetric cryptosystems, we hope that it will serve as a stepping stone toward treating active adversaries and other cryptographic primitives. Also, one should generalise the results to other formal models for cryptographic protocols (such as CSP).

Imperative program transformation by rewriting

David Lacey

Traditional rewrite systems are good for expressing transformations because:

- Rewrites specify program transformations in a very succinct manner.
- The represented transformation is intuitively expressed. Rewrite patterns express the required partial structure of transformable terms in an explicit way.
- There are existing approaches to combining rewrite sets via strategies and reasoning about their properties such as confluence, soundness and termination.

Expressing program transformations as rewrites is not a new idea, but its success has mainly been in the domain of functional programs. However, rewriting imperative programs is problematic. One of the main reasons for the difficulty is that the conditions under which it is possible to apply a rewrite are hard to describe. Many rewrite systems work by transforming abstract syntax trees and to decide whether to apply a rewrite requires complex analysis on these trees. The conditions are easier to specify if the program is represented by a control flow graph. The problem then becomes that it is not obvious how to perform rewriting on a graph.

I present a solution that tackles these problems in two ways. Firstly, a pattern matching language is developed that detects patterns in programs represented by their control flow graph. Secondly, a language is developed for specifying applicability conditions on rewrites using temporal logic constructs for reasoning over the graph. This leads to a language which is capable of expressing many common transformations on imperative programs found in optimising compilers. To give an impression of this approach, here is the specification of the optimising transformation *constant propagation*. It says that an assignment $x := v$ (where v is a variable) can be replaced by $x := c$ if the "last assignment" to v was $v := c$ (where c is a constant literal).

$$\begin{array}{ll} & n : (x := v) \implies x := c \\ \text{if} & \\ & n \vdash A^{\triangle}(\neg \mathit{def}(v)\ U\ \mathit{def}(v) \wedge \mathit{stmt}(v := c)) \\ & \mathit{conlit}(c) \end{array}$$

The side condition formalises the notion of "last assignment". It does this by stating the condition that: From node n, along all paths of computation, looking backwards, we do not define v until a point where we define v by the statement $v := c$. A break-down of how this maps to the formal language in the side condition is given in Figure 2.

Several transformations can be expressed including *dead code elimination*, *loop strength reduction* and *common sub-expression elimination* amongst others. The rewrite language has several important properties:

- The specification is in the form of a rewrite system with the advantages of succinctness and intuitiveness mentioned above.

- The rewrite system works over a control flow graph representation of the program. It does this by identifying and manipulating *graph blocks* which are based on the idea of basic blocks but with finer granularity.

- The rewrites are executable. An implementation exists to automatically determine when the rewrite applies and to perform the transformation just from the specification.

$n \vdash$ From node n,	A along all paths of computation,	$\triangle$ looking backwards
$(\neg def(v)$ we do not define v	U until	$def(v)$ we define v
$\wedge\ stmt(v := c))$ by $v := c$		

Figure 2: A break-down of the side condition

On Inverting Functions

Shin-Cheng Mu

Consider the following datatype *Tree* for a tip-valued binary tree.

$$\textbf{data}\ \mathit{Tree}\ a \quad = \quad \mathit{Tip}\ a \mid \mathit{Bin}\,(\mathit{Tree}\ a)\,(\mathit{Tree}\ a)$$

Given a list of integers, say $[3,3,3,4,4,3,3,2]$, representing the depths of the leaves of a tree from the left to the right, how can we rebuild (the shape) of the tree, as in Figure 3?

The problem can be specified in terms of the inverse of the function $\mathit{flatten} :: \mathit{Tree}\ a \rightarrow [a]$, which takes a tree and returns its tips in left-to-right order:

$$\mathit{build} \quad = \quad \mathit{ok} \cdot \mathit{flatten}^{\circ}$$

In the above specification, $\mathit{flatten}^{\circ} :: [a] \rightarrow \mathit{Tree}\ a$ denotes the inverse of *flatten* and $\mathit{ok} :: \mathit{Tree}\ \mathit{Int} \rightarrow \mathit{Tree}\ \mathit{Int}$ is a partial function acting like a filter which maps a tree to itself only if all the values on its tips equal their depths.

In fact, many problems in computation can be specified in terms of computing the converse of an easily constructed function. The purpose of this talk is to describe one technique for inverting functions and to illustrate it with three examples. All three problems involve building (or rebuilding) a tree of some kind, and all can be specified in terms of the converse operation of flattening a tree into a list of its values.

Functional programmers are aware that flattening a structure is usually performed by a fold operation. Consequently, building a structure is usually performed by the converse operation of unfolding. However, this is not necessarily so. The converse of a function theorem, to which this talk is devoted, gives us conditions under which the inverse of a function can be written as a fold. The theorem, to my knowledge, was first proposed by de Moor [2]. In this talk we will talk about a special instance of the theorem on lists. The more general theorem and its proof can be found in [1].

The inverse of a function is not in general a function. Taking $\mathit{flatten}^{\circ}$ above for example, there are many trees which could flatten to the same given list. Therefore we will generalise to relations, allowing a value in the domain to be mapped to multiple results in the range. The fold delivered by the converse of a function theorem is usually a relation. It is then composed with a filter like *ok* choosing among the trees those satisfying some criteria.

The next step is to apply the well-known fold fusion theorem to fuse the filter into the fold to remove the nondeterminism in the fold, thus refining the specification to an implementable function.

Continuing with the example, applying the converse of a function theorem to $\mathit{flattenF} :: [\mathit{Tree}\ a] \rightarrow [a]$, the function flattening a forest, we can prove that $\mathit{flattenF}^{\circ}$ is a fold:

$$\begin{array}{lcl} \mathit{flattenF}^{\circ} & = & \mathit{foldr}\ \mathit{add}\,[\,] \\ \mathit{add}\ a\,(us \mathbin{+\!\!+} vs) & = & [\mathit{foldl}\ \mathit{Bin}\,(\mathit{Tip}\ a)\ us] \mathbin{+\!\!+} vs \end{array}$$

which is a relation because we made use of a nondetermistic pattern matching in *add*. A

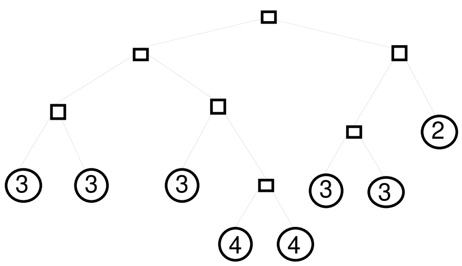

Figure 3: A tree whose tips have depths $[3, 3, 3, 4, 4, 3, 3, 2]$

functional refinement of *build* can then be derived as outlined:

$$
\begin{aligned}
build \quad = & \quad \{flatten = flattenF \cdot wrap \text{ where } wrap\ x = [x]\} \\
& ok \cdot wrap^\circ \cdot flattenF^\circ \\
= & \quad \{\text{invent } ok' \text{ such that } ok \cdot wrap^\circ = wrap^\circ \cdot ok'\} \\
& wrap^\circ \cdot ok' \cdot flattenF^\circ \\
= & \quad \{\text{fold fusion theorem}\} \\
& wrap^\circ \cdot foldr\ add'\ [\,]
\end{aligned}
$$

where add', having being fused with ok', is a function, thus implementable in Haskell. The other two problems can also be solved in a similar fashion.

Functional programmers make use of a handful of laws and theorems to transform specifications to optimised code. Among them, the fold fusion theorem has proved to be one of the most important. The converse of a function theorem is another useful tool worth being added to the functional programmer's arsenal. Its joint use with the fold fusion theorem turns out to be a recurring pattern in program derivation.

References

[1] Richard Bird, Oege de Moor, *Algebra of Programming*, Prentice Hall, 1997.

[2] Oege de Moor, Jeremy Gibbons, Pointwise Relational Programming, *Proceedings of Algebraic Methodology and Software Technology 2000* Lecture Notes in Computer Science 1816, pages 371-390, 2000.

Machine-independent characterizations of complexity classes
(a survey)

Andrzej S. Murawski

Complexity classes were originally defined using machine models and restrictions on either their runtime or space consumption. These low-level descriptions might have led to suspicion that complexity theory would be confined to combinatorial manipulations with Turing machines yielding more abstract methods ineffectual. However, it turned out that many of the definitions are robust and do not depend on technical details, like the number of tapes or the size of the alphabet. They also coincided for various machine models (Turing machines, random access machines, pointer machines etc) — a clear indication that a deeper concept is involved. Furthermore, for a large number of complexity classes, machine-independent characterizations have been discovered bringing complexity within range of alternative investigative methods. A lot of open problems could then be recast into novel settings posing challenging problems in other theories, often central to them.

Logicians were the first to contribute to that area and initiated a new discipline, now called descriptive complexity [1]. It does not come as a surprise that logics can be used to specify decision problems. Such an approach offers new ways of calibrating expressivity by the quantifier pattern, number of variables used or the length of formulas. No encodings are involved, which is a definite advantage over the machine-dependent representation. Moreover, it seems plausible that properties that are harder to check will be harder to express. The accuracy of that correspondence exceeds all expectations – restrictions placed on the syntactic parameters "produce" standard complexity classes in a resource-free way.

Assuming that graphs are represented by pairs $\langle V, E \rangle$ where $E \subseteq V \times V$, we can express the well known NP-complete problem 3-COLOURABILITY in second-order logic as follows:

$$\begin{aligned}\exists R \exists G \exists B \cdot \forall x \forall y \cdot \quad & [(E(x,y) \Rightarrow \neg(R(x) \wedge R(y)) \wedge \neg(G(x) \wedge G(y)) \wedge \neg(B(x) \wedge B(y))) \\ & \wedge (R(x) \vee G(x) \vee B(x))]\end{aligned}$$

The first clause states that no two adjacent vertices have the same colour and the second says that every vertex has some colour. In fact, any problem in NP can be defined by a formula of existential second-order logic ($SO\exists$):

$$\exists R_1 \ldots \exists R_n \cdot \phi$$

where ϕ is a first-order formula. The converse is also true: any $SO\exists$ formula defines a property which can be verified in nondeterministic polynomial time. Consequently, it can be said that existential second-order logic characterizes precisely the problems in NP. What is the logical characterization of polynomial-time computability then and how is it different from that of NP? Surprisingly, it suffices to restrict the shape of ϕ to a (universally quantified) conjunction of Horn clauses and one gets an analogous correspondence, this time with PTIME. Note that the formula used for 3-COLOURABILITY does not consist of Horn clauses. Otherwise, the results mentioned so far would imply a positive answer to the P=NP problem. The formula below is Horn and specifies the unreachability of a vertex y from x in a graph, a problem known to be in P:

$$\exists R \forall a \forall b \forall c \cdot [R(a,a) \wedge (E(a,b) \Rightarrow R(a,b)) \wedge (R(a,b) \wedge R(b,c) \Rightarrow R(a,c)) \wedge \neg R(x,y)] \cdot$$

An alternative logical link with PTIME can be established via first-order logic with inductive definitions. For example, the reachability of y from x is expressed by

$$R(x, y) = E(x, y) \vee \exists z \cdot (R(x, z) \wedge R(z, y))\cdot$$

Similarly, the properties expressible in fixpoint first-order logic are exactly the PTIME problems. In addition, there exist equally sastisfying descriptions of LOGSPACE, NLOGSPACE, PSPACE and many other complexity classes.

Not only can these characterizations be used to infer the complexity of problems given their specification, but they can also suggest programming languages that capture complexity classes. The connexion with logic programming is apparent, but other programming paradigms have also benefited from descriptive complexity. The crucial observation allowing to control the complexity of functional and imperative programs is that no constructors are present in the logical specifications (e.g. binary strings are represented by $\langle\, \{0, 1, \ldots, n\}, R, succ\, \rangle$ where $R \subseteq \{0, 1, \ldots, n\}$, $R(i)$ holds iff the ith bit is to be 1 and $succ$ is the restricted successor function on $\{0, 1, \ldots, n\}$, and consequently there is no provision for the creation of new values that could be used in auxiliary definitions). An implementation of the idea would be a functional programming language with binary lists and booleans as ground types, product types and function definitions with first-order arguments. The head and tail functions would be allowed, but not Cons. It has been shown that one can solve any PTIME problem using this language and, conversely, the semantics of each Cons-free program of type $[Bool] \rightarrow Bool$ (provided it terminates for any input) is a problem in P [3]. One may force termination by using primitive recursion, but then the Cons-free first-order programs solve problems in LOGSPACE. The question whether LOGSPACE = PTIME is still open, but in this setting it boils down to the relationship between primitive and general recursion. Another interesting fact is that primitive recursion captures PTIME again if second-order values are permitted as arguments. Yet another increase in the order of arguments results in PSPACE.

We cannot consider Cons-free languages if we want to compute PTIME numeric functions as opposed to decision problems. Cons must then be added to the language, but this makes even primitive recursion too powerful. To achieve polynomial time complexity, further restrictions on recursion are necessary. It helps to observe that the argument of the step function in recursive schemes which is used to process recursive calls cannot be recursive itself. An abstract presentation of that idea introduces levels for inputs and output. Thus, the type of recursive arguments ought to be declared higher than that of the relevant arguments of step functions. A synthesis of this approach leads to a framework consisting of composition and recursion schemes respecting the stratification. The functions definable in that system are precisely the polynomial time computable numeric functions [2].

References

[1] N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.

[2] D. Leivant *Stratified functional programs and computational complexity*. In Proc. 20th ACM POPL, pages 325–333. ACM Press, 1993.

[3] N. D. Jones *LOGSPACE and PTIME characterized by programming languages.* Theoretical Computer Science, 228: 151–174, 1999.

Transforming Abstract Data Types by Rewriting

Iván Sanabria-Piretti

Abstract data types play a significant role in the development of reliable software. In the specification phase, designers use abstract data types in their algorithms as a tool to reduce the amount of complexity or detail that must be considered at any one time. This is because algorithms described in terms of abstract data types are easier to write, understand and communicate. It is also simpler to reason about their correctness, because the possible representations to be used in their implementation, as well as any algorithmic complexity exclusively associated to these representations, are abstracted away. Decisions about the implementation of the abstract data types are made in the second phase in the development process, the implementation phase. In this phase, implementers select the most suitable representation for each abstract data type in the context of the whole specification. In this talk, I will show how to make this phase easier with the help of rewriting tools in the context of a basic imperative language. This work is motivated by previous research done in other projects like [1, 2, 3].

During the implementation phase, implementers have to commit each of the abstract data types in the specification to a concrete representation that satisfies the requirements. However, committing to specific representations without full awareness of the context may not be convenient. For instance, let us consider a simple, common scenario. Most of us have been confronted with the problem of implementing a program from a specification found in some textbook. Most likely, the specification was written in terms of abstract data types, where their implementations were left as an exercise at the end of the chapter. Once we had struggled to get our implementation right, we remembered seeing a pair of lines at the bottom of the specification suggesting changes to these abstract data types in order to obtain better results. However, such changes would have dramatically impacted our implementation. Wouldn't it be nice if we could make such changes at the specification level and then allow a transformation system to assist us to do the job of generating the implementation again?

In this talk, I describe a data type transformation system I am currently developing. In my solution, the programmer adds his intentions to the specification in a precise and concise way, and the intentions then guide the transformation system towards the desired implementation. If the result doesn't satisfy his needs, the programmer can go back to the specification and change the initial intentions. As a motivation example, I present a specification for the construction of Huffman trees, where the declaration of the abstract variable S of type $PQ\ Htree$, a priority queue of huffman-trees, is annotated as follows:

$$\underline{var}\ S :: (PQ \Rightarrow List; List(Htree \Rightarrow HRec); List \Rightarrow LRec; LRec\_HRec \Rightarrow LHRec)\cdot$$

The annotation states that the type of S will be represented as the type *LHRec*. This final representation of S is obtained after applying a sequence of data type transformations. $A \Rightarrow B$ stands for a data type transformation from the type A to the type B. First, $PQ \Rightarrow List$ transforms S from a $PQ\ \alpha$ into a $List\ \alpha$, then the functorial transformation $List(Htree \Rightarrow HRec)$ transforms the element type of the list from *Htree* to the record representation *HRec*. Next, $List \Rightarrow LRec$ transforms the $List\ \alpha$ into a record representation

LRec α. At this point, the representation of S is given in terms of a data structure composed by record types, *LRec HRec*. Finally, a further transformation $LRec\_HRec \Rightarrow LHRec$ merges these two record types into a single one, *LHRec*, creating a multi-linked record structure, which is a compact structure usually found in hand-coded implementations. An important observation of this sequential process is that more implementation details are exposed with each transformation step.

In my view, an abstract data type may be transformed directly into one of its final implementations in a single step. However, in the case of more complex data types, a series of transformation steps may be needed, each producing intermediate data structures that provide more insights about a final implementation. It is also my belief that if smaller, simpler transformations are applied on the intermediate results, then it is more likely that local optimizations may take place, hence contributing to a better final result.

References

[1] F.L. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*. Lecture Notes in Computer Science ; 292. Springer-Verlag, Berlin; New York, 1987.

[2] D. Gries and D. Volpano. The transform - a new language construct. *Structured Programming*, 11:1 – 10, 1990.

[3] Y. Smaragdakis and D. Batory. Distil: a transformation library for data structures. In *Proceedings of the Conference On Domain-Specific Languages*, pages 257–269, Santa Barbara, CA, 1997. Usenix Association.

Re-entrant locks in Java

Silvija Seres

based on work with Raymie Stata,
Rustan Leino and Greg Nelson at Compaq SRC

In Java, a thread can acquire the same lock a multiple number of times, and it has to release it the same number of times before another thread can acquire it. Such locks are known as "thread re-entrant monitors," as against the "non-re-entrant" locks found in languages like Modula-3 where a thread will block forever if it attempts to acquire a lock it already holds. In this talk we summarise the two approaches and discuss why we believe that Java's approach is not optimal.

To keep Modula-3 threads from deadlocking with themselves, one would require that the call `Thread.Acquire(m)` has as a precondition that $m \notin LL$ (where LL is the set of locks held by the current thread). One can go further in trying to prevent a thread from deadlocking with *other* threads. The standard approach to deadlock avoidance is to define a partial order on locks and to require that threads acquire locks according to this order. This leads to a stronger precondition for `Thread.Acquire(m)`: $Sup(LL) < m$ (meaning "the 'largest' lock held by the current thread is smaller than m").

Java does not have routines analogous to `Thread.Acquire` and `.Release`, but only has a `synchronized` statement that matches a lock acquisition with a release. The precondition for a deadlock-free `synchronized` block can be slightly more liberal than the precondition outlined above, because with Java's thread re-entrant locks a thread can not deadlock with itself. Three possible precondition patterns emerge:

1. $Sup(LL) < m \vee m \in LL$
2. $Sup(LL) \leq m$
3. $Sup(LL) < m$ (this is the Modula-3 pattern).

Preconditions patterned after (1) allow routines to be called under the widest set of circumstances. While the other two patterns rule out callers who hold a lock greater than m in the locking order, (1) does not. The trade-off, however, is that routines with preconditions patterned after (1) must explicitly mention every re-entrant lock (such as m) the routine may want to acquire. This is often too much information, revealing locks that should be hidden from clients.

Preconditions patterned after (2) do not have this limitation: such preconditions allow a routine to lock any lock after m in the locking order and thus do not have to mention every lock that might be acquired by the routine. However, this added flexibility for the implementation comes at the expense of flexibility for clients by restricting calls to program states in which nothing greater than m has been previously acquired.

Preconditions patterned after (2) have another problem, a problem shared by preconditions patterned after (1): they make it impossible to assume monitor invariants inside the bodies of routines. A monitor invariant is a relation among the variables protected by the monitor. Threads establish a lock's monitor invariant before they leave the monitor; this means that a thread entering the monitor may assume the monitor invariant. However, with thread re-entrant locks, a thread can only assume a monitor invariant when it enters the monitor for the "first time" (ie, whenever the thread increments the lock's count from zero to one). When the thread re-enters the monitor, there is no guarantee that the monitor's invariant has been re-established, so it cannot be assumed. Connecting this back to preconditions, preconditions patterned after (1) and (2) allow $m \in LL$ on entry to a routine; this means that when the routine acquires m, it might not be doing so for the first time, so it cannot assume the monitor invariant.

Preconditions patterned after (3) do not have the limitations of either (1) or (2): they allow routines to acquire locks in addition to m, and they allow routines to assume the monitor invariant associated with m when m is acquired. However, preconditions patterned after (3) are the most restrictive in terms of the program states in which they allow routines to be called: they restrict calls to program states in which nothing greater than m nor m itself have been locked. When preconditions patterned after (3) are used, the re-entrant property of monitors is not used; this explains why, as noted above, pattern (3) is the pattern used in Modula-3.

Thread re-entrant locks seem like a convenience at first. For example, with non-re-entrant locks, it is often necessary to write two versions of a routine, one that grabs the lock first and another that assumes the lock has already been grabbed. With re-entrant locks, only one version is needed. Also, with non-re-entrant locks, one is often faced with a routine R that calls another routine R' that eventually (through a long series of calls) wants to acquire a lock already acquired by the original routine R. The fix for these situations often requires some awkward coding. With thread re-entrant locks, R' is allowed to reacquire the lock so no awkward fix is required.

These and other situations lead people to advocate re-entrant locks. However, although re-entrant locks can be a convenience, they turn out to cause more trouble than they are worth. When locks are *not* re-entrant, errors lead to deadlocks, which are relatively easy to find. When locks are re-entrant, it is easy to lose track of where one can and cannot assume the monitor invariant, and invariant errors lead to corrupt state that might not manifest itself visibly until program execution has wandered far from the actual error, making errors difficult to find.

The difficulties associated with thread re-entrant locks suggest that Java programs should avoid using the re-entrant property of locks. This means that preconditions patterned after (3) should be preferred.

Annotating programs for efficiency

Ganesh Sittampalam

Introduction

Consider the following Haskell program for calculating the minimum depth of a leaf-labelled binary tree.

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

$$\begin{aligned}
&\mathit{mindepth}\ (\mathit{Leaf}\ x) &&= 0\\
&\mathit{mindepth}\ (\mathit{Node}\ s\ t) &&= 1 + \mathit{min}\ (\mathit{mindepth}\ s)\ (\mathit{mindepth}\ t)
\end{aligned}$$

The program is short, clearly stated, and it is easy to see that it will do what we intend. However, it is not very efficient. Imagine a right-leaning tree; our program will quickly explore the left branch of the tree, but will then spend a significant amount of time in the right-hand branch despite the fact that it will quite soon become apparent that the leaf of minimum depth must be in the left branch and that the search of the right branch could be aborted.

A little bit of thought allows an experienced programmer to make this definition more efficient; simply add two accumulating parameters to keep track of the minimum depth found so far in the tree and the current depth, and then the search can be cut off for any particular branch if the current depth reaches the minimum already found:

$$\begin{aligned}
&\mathit{fastmd}\ (\mathit{Leaf}\ x)\ d\ m &&= \mathit{min}\ d\ m\\
&\mathit{fastmd}\ (\mathit{Node}\ s\ t)\ d\ m &&= \textbf{if}\ \mathit{nd} \geq m\ \textbf{then}\ m\\
& && \qquad\qquad\quad \textbf{else}\ \mathit{fastmd}\ s\ \mathit{nd}\ (\mathit{fastmd}\ t\ \mathit{nd}\ m)\\
&\quad \textbf{where}\ \mathit{nd} = d + 1
\end{aligned}$$

$$\mathit{mindepth}\ t = \mathit{fastmd}\ t\ 0\ \infty$$

However, it is not nearly so obvious that the resulting program is correct, and it is certainly much harder to understand than the original code, which increases the likelihood of bugs and makes the code harder to maintain. What we need is a programming paradigm that combines the best of both worlds – the clarity of the first program and the efficiency of the second.

The key point to note here is that the first and second programs can be related by equational reasoning, and that in fact once we have made the insight that we should use "minimum depth found so far" and "current depth" as accumulating parameters, deriving the second program from the first is quite straightforward.

Thus, it makes sense to give the original program with something like the following annotation:

$$\begin{aligned}
&\textbf{transform}\ \mathit{mindepth}\ t = \mathit{fastmd}\ t\ 0\ \infty\\
&\textbf{where}\ \mathit{fastmd}\ t\ d\ m = \mathit{min}\ (\mathit{mindepth}\ t + d)\ m
\end{aligned}$$

In other words, we have redefined *mindepth* in terms of the new function *fastmd*, and shown how *fastmd* can be defined from the original definition of *mindepth*. What we now expect is that our compiler will be able to eliminate *mindepth* from this definition and derive the efficient form for *fastmd* given above.

However, automatic program derivations are notoriously unpredictable and difficult for a compiler to direct without specialist knowledge provided by the programmer. Since we would like to have confidence in what our compiler will do, we should make the effort to provide such knowledge to it.

Fusion

It turns out that the *fold-fusion* rule provides exactly the framework we need. This rule tells us that under certain conditions a recursive traversal of an inductive datatype followed by another function can be fused into just a recursive traversal. The version of this rule for binary trees is as follows:

$$\begin{array}{rcl} f\ (\mathit{foldbtree}\ n\ l\ t) & = & \mathit{foldbtree}\ n'\ l'\ t \\ \text{if} & & \forall a . l'\ a = f\ (l\ a) \\ & & \forall b\ c . n'\ (f\ b)\ (f\ c) = f\ (n\ b\ c) \end{array}$$

The function *foldbtree* is the "natural" fold on binary trees, which encapsulates recursive traversals.

$$\begin{array}{ll} \mathit{foldbtree}\ n\ l\ (\mathit{Leaf}\ a) & = l\ a \\ \mathit{foldbtree}\ n\ l\ (\mathit{Node}\ s\ t) & = n\ (\mathit{foldbtree}\ n\ l\ s)\ (\mathit{foldbtree}\ n\ l\ t) \end{array}$$

Our program for *mindepth* is not stated in terms of *foldbtree*, and thus it seems that the fusion rule is not immediately applicable. However, we find that we can introduce the "identity" fold function *foldbtree Node Leaf* by applying it to the binary tree parameter t in the definition of *fastmd*, thus avoiding the need to manually express *mindepth* in an appropriate form. In addition, the result of applying the fusion rule is also expressed in terms of *foldbtree*, and in the interests of producing a readable program it makes sense to inline the definition of *foldbtree* afterwards.

Applying the fusion rule automatically requires the synthesis of completely new definitions for functions, such as the n' in the version of the fusion rule given above; we have developed some novel *higher-order matching* algorithms which do this in the course of applying the fusion rule.

Thus, as well as the annotation given above we add the hint that fusion should be used on the parameter t. The final pieces of information we find that we need to add are some algebraic laws about functions like *min*, which are required in order to satisfy the side conditions in the fusion rule. Given this information, our system MAG is able to carry out just this kind of derivation for *mindepth* and other examples such as alpha-beta trees and the Fibonacci sequence.

Reasoning about quantum-mechanical nonlocality

Paolo Zuliani

In this talk we shall show how it is possible to use a suitable programming language ($qGCL$) for reasoning about one of the strangest properties of quantum mechanics: nonlocality.

Nonlocality is the fact that an event occuring in one point of the universe can have a *simultaneous* effect in another part of the universe. The two parties may be arbitrarily distant, such that no signal can usefully travel between them.

The aim of this work is two-fold: to provide a further test for the suitability of $qGCL$ as a tool for describing and reasoning about quantum systems and thereby providing a rigorous treatment within Computation of various *Gedankenexperimente* from Quantum Physics.

$qGCL$ [6] is a general-purpose programming language for quantum computation developed as a superset of the *probabilistic* Guarded Command Language ($pGCL$). In particular, $qGCL$ extends $pGCL$ with four constructs:

- transformation q, that converts a classical bit register to its quantum analogue, a *qureg*;
- *initialisation*, which prepares a qureg for a quantum computation;
- *evolution*, which consists of iteration of unitary operators on quregs;
- *finalisation* or observation, which reads the content of a qureg.

$qGCL$ enjoys the same features of $pGCL$: it has a rigorous semantics and an associated refinement calculus, which include program refinement, data refinement and combination of specifications with code. $qGCL$ has been successfully used to describe and reason about all known quantum algorithms and also to derive one of them (the Deutsch-Jozsa algorithm) from its specification.

The most famous example of nonlocality is the so called EPR paradox, named after a paper by Einstein, Podolsky and Rosen in 1935 [3]. Their Gedankenexperiment consists in a source emitting two spin-$\frac{1}{2}$ particles, whose total spin is zero. The two particles move away one from the other and the components of their spins are subsequently measured in the same direction by two observers, P and J say. Conservation of total spin entails that if P measures $+\frac{1}{2}$, then J will necessarily measure spin $-\frac{1}{2}$ and *vice versa*. The results are thus *anticorrelated*. The paradox lies in the fact that whoever between P and J measures the spin first, the outcome of the measurement will *simultaneously* constrain the measurement of the other party, no matter how far it is. Using $qGCL$ we shall write a program to formalize this paradox.

Nonlocality is an unavoidable feature of quantum mechanics, as John Bell proved in 1964 [2]. He developed a set of inequalities which are satisfied in any local physical theory consistent with quantum theory results, except quantum mechanics itself. Experiments performed by Alain Aspect *et al.* [1] in the 80's confirmed the violation of Bell's inequalities, in the way predicted by quantum theory.

Since Bell's original work, many other examples of 'Bell inequalities' have been developed. We will here consider two examples by David Mermin [4, 5]: the former is on the same line of Bell's work, describing a Gedankenexperiment which offer a statistical violation of some properties true in any local physical theory (statistical violation means that the properties are refuted only over a series of runs of the experiment). The latter offers a more dramatic violation of locality: a *single* run of his Gedankenexperiment suffices to rule out local theories whishing to mimic quantum theory results.

For both examples we shall write the corrisponding *qGCL* program simulating the experiments; we shall also write the programs implementing the local theories constrained to reproduce the same results of quantum theory. Using standard expectation-transformer semantics we will compare the quantum and the local programs on selected expectations, thus deriving the desired violation of locality.

References

[1] A. Aspect, P. Graingier, and G. Roger. Experimental realization of EPR Gedankenexperiment: A new violation of Bell's inequalities. *Physical Review Letters*, 49:91–94, 1982.

[2] J.S. Bell. On the Einstein-Podolsky-Rosen paradox. *Physics*, 1:195–200, 1964.

[3] A. Einstein, B. Podolsky, and N. Rosen. Can quantum mechanical description of physical reality be considered complete? *Physical Review*, 47:777–780, 1935.

[4] N. David Mermin. Bringing home the atomic world: Quantum mysteries for anybody. *American Journal of Physics*, 49:940–943, 1981.

[5] N. David Mermin. Quantum mysteries revisited. *American Journal of Physics*, 58:731–734, 1990.

[6] J.W. Sanders and Paolo Zuliani. Quantum programming. *Mathematics of Program Construction, Springer LNCS*, 1837:80–99, 2000.