Pattern Transfer: bridging the gap between theory and practice

Nick Benton
Microsoft Research
Cambridge
nick@microsoft.com

1 Introduction

Universities and industrial research laboratories around the world contain hundreds of professionals who would describe themselves, or at least admit to being, theoretical computer scientists. They certainly seem to have a lot of fun doing whatever it is that they do, but those who regard 'theoretical' as the antonym of 'practical' sometimes ask whether any of it is of interest to a broader community, or more narrowly, whether it has any industrial relevance. In this essay I will attempt to argue (rather unsurprisingly) that the answer to both of these questions is 'yes', and to discuss how theory and practice should interact.

The discussion is, of necessity, mostly based on the kind of theory and the kind of practice of which I have some direct experience, so I should start by clarifying what I mean by 'theory'. It seems reasonable to identify as a community those computer scientists, mathematicians and logicians who do research in or around the areas of logic and type theory, semantics of programming languages, functional programming, concurrency theory, automated reasoning and logics for reasoning about hardware and software systems. This kind of research is sometimes called 'Eurotheory' because, although its practitioners are spread all over the world, it seems to be regarded as slightly more mainstream in Europe than it is in North America (where 'theoretical computer science' often implies work on algorithms and complexity theory). For the present discussion, I will use the term 'theory' in its European sense.

This collection of research areas is certainly very broad, ranging from fairly abstract category theory to the implementation of programming languages and theorem provers. Nevertheless, the fields do form a fairly coherent whole: they share a lot of common concepts, techniques and tools and there are many individuals who have made significant contributions to all of them.

Work in theoretical computer science tends to be mathematical (or at least formal) in nature, to involve a close analysis of the fundamental structures arising in computer science, to emphasize correctness, proof and reasoning principles, to be at a high level of abstraction and to proceed by investigation of paradigmatic foundational systems such as various λ-calculi or the π-calculus.

Theory is widely accepted as a major part of computer science. It is a very active research area, as witnessed by the large number of journals and conferences devoted to it, and also accounts for much of the undergraduate curriculum at most leading universities. Theoretical computer science is a fascinating and exhilarating field, with great cultural and intellectual value. From Turing's work on computability through the Curry-Howard correspondence, Scott's domain theory, Milner's CCS and Hoare's CSP to the linear logic of Girard, categorical logic and game semantics, theoretical computer science has produced much that is beautiful and deep. Mathematical logic has blossomed under the influence of computer science, which has given a new significance to areas such as constructive logic and proof theory (which even most mathematicians would have regarded as marginal a few decades ago). The 'big questions' of theory – What is the relationship between the declarative and the procedural? How should we describe, structure and reason about complex interacting systems? – have an importance which transcends their origins in particular engineering problems. The notions of information and computation are, along with relativity, quantum mechanics, evolution and genetics, amongst the defining scientific ideas of our times.

In spite of this heady subject matter, theoretical computer science is not a branch of philosophy, nor even of pure mathematics. The thing that keeps theoretical computer science focussed, vital and (let us not forget) comparatively well-funded, is its relation to computing practice.

The influence of practice on theory is not hard to see. The great pace of innovation in the computer industry provides a continual stream of new ideas and problems for theoreticians to investigate and a large collection of concrete examples against which abstract theories can be tested. But the traffic is supposed to be two-way – theoretical research is claimed to have application to the design of new languages, systems and methodologies, to the specification and verification of computer systems, and to the analysis and synthesis of both software and hardware. In this respect the situation appears less satisfactory: most of the computing industry *seems* to work and innovate quite successfully, hardly ever worrying at all about the issues which are the daily concern of theoreticians. Of course, there are examples of successful applications of ideas and results originating in the theory community, some of which are discussed below, but few would argue that these are typical of mainstream industrial practice.

Is this true; if so, is it a problem and should we care? Amongst the positions one could take are four that could be caricatured as:

Arrogant ivory tower: "Who cares? Let's just get on with doing intellectually satisfying abstract work. If we can get funded for it because somebody thinks there might be industrial applications then that's fine, but it's not our problem."

Refined ivory tower: "It's inevitable that basic scientific work is often a very long way from many of the applications for which it, in theory, provides foundations. Theory and practice are, after all, different things and attempts to manage their interaction or to force them closer together are detrimental to both.

"If there is a certain amount of tension between the theoretical and the applied in computer science, it's surely no greater than in many other disciplines, such as economics, chemistry or biology. We shouldn't encourage division, but neither should we be too concerned about it.

"In many areas, fundamental science *has* proved to have useful application, and this is one major justification for doing it. But the application is often much later and may even be to a problem that could not have been envisaged when the original research was done. Furthermore, worrying about 'the mainstream' is misguided – it is only to be expected that practical applications of theory will commonly be in fairly specialised domains.

"To make an extreme analogy, it seems unlikely that many theoretical physicists spend much time worrying about how to get the results of their research into the civil engineering community. This is despite the fact that the fundamental natural laws being investigated by the physicists 'explain', say, the properties of materials of interest to somebody designing a bridge. The gap between the levels of description is so vast that that kind of theory is of no use in solving the problems arising in that kind of application. Indeed, in this case the gap is so large that it is inhabited by at least two other disciplines – chemistry and materials science – each with their own distinct kind of theory appropriate to their particular levels of abstraction.

"So whilst we should keep an eye out for the odd application which is clearly close to the theory, the main thing is just to do good science, confident that some of it will prove to be of practical benefit eventually."

Unappreciated genius: "Most computer programs are ugly messes, designed in an unprincipled way and implemented using stone-age languages by ill-educated hackers. Of course nothing ever works properly. If they'd just use formal logic x and programming language y, then everything would be fine. But they never listen."

Hard-nosed real-worlder: "Computing is a practical engineering discipline, not a branch of mathematics. Most theory is just a sign of 'physics envy', and contributes nothing to the solution of the problems of building real computer systems. All the foundations we need, such as automata theory and computability, have been well understood for decades. Insofar as there is scope for further theoretical research, it should be driven by the immediate needs of engineers and industrialists." (By contrast with the others, this is a view more likely to be held by a non-theorist, but it is also consistent with being an unusually cynical ivory-tower dweller, or a disillusioned former unappreciated genius.)

In this article I will attempt to argue for a more flexible attitude, which tries to combine a confidence in the relevance and importance of abstract foundational research with a willingness to tackle practical problems.

2 The applicability of theory

In computing, the real conceptual distance between the most abstract theory and the screen-face of practical development work is far smaller than in the physical sciences and engineering, firstly because computing is still a fairly young subject and secondly because writing code or designing hardware is inherently a highly abstract activity compared with most engineering tasks. The 'real worlder' sees the gap as large and unbridgable (or, worse, not worth bridging), and I believe that he is entirely wrong. The 'refined ivory tower-dweller' also sees the gap as large and has no personal interest in trying to cross it; instead, she imagines communication between the two sides being achieved by 'Chinese whispers' passing along a chain of people ranging from the most applied on one side to the most theoretical on the other. But the whole point of the game of Chinese whispers is that it is an unreliable means of communication, and poor communication is a large part of the gap which many see between theory and practice in computing. It is far better for individuals to work on both sides whilst that is still feasible; that way we may prevent the gap becoming a gulf.

The 'arrogant ivory tower-dweller' simply risks missing out, not only on potentially exciting and satisfying application-related work, but also on a whole range of new perspectives and intuitions which applications can bring to theoretical work.

Foundational research in computer science has much more to offer computing practice than is often realised, although the explicit contribution will only occasionally be in the form of an elegant collection of independently-developed definitions and theorems. It is easy to think that, because theoretical research often involves precise, even pedantic, analysis of computational structures (e.g. in deriving sound principles for proving the correctness of programs), it is only of benefit in application areas where such complex and expensive formal reasoning is both feasible and justifiable. This seems to me to be mistaken.

It is certainly true that theory-based formal methods have had a significant effect on practice in areas such as safety-critical systems, security and hardware design, in which the potential cost of bugs is extremely high. These are the obvious, high-profile application areas for theory: those in which the effort of developing a formal specification of the problem and a proof of correctness of the solution pays for itself in increased confidence in the resulting system. Theory's potential for continued contribution to this sort of work is clear: firstly because the demand for provably correct systems can only increase as information technology plays an ever greater part in most people's lives, and secondly because further developments in the underlying technology of formal methods (both foundations and tools) will make them viable in a wider range of applications.

Without wishing in any way to minimise the importance of formal methods, however, it should be emphasised that much of the mutually beneficial interaction between theory and practice actually takes place via a more informal exchange of concepts, idioms and folklore. This is supplemented by occasional nuggets of more rigorously developed theory being applied to relatively small components of larger systems, and by increased use of languages and tools whose design has been influenced by theoretical work (of which the users are typically unaware). This might sound alarmingly vague and woolly, particularly to a formally-inclined theorist, but it is a much more realistic view than that of our 'unappreciated genius' and provides a much wider range of opportunities for theory to influence practice. It seems improbable that there is a grand unified theory of programming which could solve all the problems of software development, but theorists *have* already developed a large collection of ideas and ways of thinking which can be usefully applied, albeit sometimes in a slightly 'impure' form, to solving practical problems today. Encouragingly, the last few years have seen a noticable increase both in real applications of ideas originating in the theory community, and in the development of new theories for specific contemporary problem areas. For example:

Year 2000: AnnoDomini is a commercial source-to-source analysis and transformation tool for solving Year 2000 problems in Cobol programs [16]. It is written in Standard ML and uses a novel type inference system both to detect potential Year 2000 bugs and to control program transformations. The underlying type theory, whilst non-trivial and specific to the problem, is based on ideas of type inference, polymorphism and subtyping, which have been developed by the theory community over the last twenty years to the point where they are now fairly standard.

Java: The success of the Java programming language [4] has motivated much recent theoretical work. Java has increased general interest in programming languages and popularised many features which were hitherto mostly confined to non-mainstream languages popular with theoreticians (e.g. type-safety and automatic storage management). A feature of Java that is of particular interest to theoreticians is its use of bytecode verification: untrusted code is automatically analysed before being executed to prove that certain security invariants will not be violated. Java's type system, security model and verifier have been extensively formalised and investigated (e.g. [14, 42]), leading to the discovery of bugs (e.g. [13, 20]) and the proposal of numerous improvements and extensions (e.g. [8, 1]). It is pleasing to observe firstly that existing logical, type theoretic, semantic and theorem-proving techniques have proved effective in studying the new language, and secondly that interesting new research directions have been suggested by the experience.

Whilst a committee of academic language theorists might well have come up with a slightly different design for Java (it would certainly have included parametric polymorphism, for example), what is undeniable is that the current language draws heavily on ideas which were 'in the air' because of more theoretical research work. (So much so that it has been suggested that Java's popularity is such that advocates of functional programming should now simply declare victory!)

Ambients: Java is but one example of a much wider trend towards network-oriented computing. Specifying and enforcing security policies in an environment in which both code and machines may be mobile is extremely challenging – this is an area in which there is a clear industrial need for new ideas. Cardelli and Gordon's ambient calculus [9, 10] is an expressive formal system designed specifically to model issues of mobility and access control. Although the calculus is novel, and is intended to address new, practical questions associated with firewalls, applets and so on, it is founded in a great body of previous work from the theory community, such as the π-calculus, operational semantics, various forms of bisimulation, modal logic and type theory.

Functional programming: Functional programming languages have for decades been both objects of study for theoreticians and a favourite tool for implementing 'theoretically-motivated' applications such as theorem provers and optimising compilers. Compared with the huge number of programmers who use C, C++ or Cobol, of course, it is still almost true to say that 'no one uses functional languages' [47], but functional languages *are* becoming both more usable and more widely used.

The most commonly-cited example of the industrial use of functional programming is undoubtedly Ericsson's language Erlang [3]. The achievements of the Erlang team are extremely impressive: they have designed and implemented their own concurrent functional language, tools and libraries which are used by hundreds of programmers to develop very large telecommunications applications which are widely deployed in real products. Along the way they have gathered real evidence for the benefits of functional programming and overcome many technical (and at least as many non-technical) obstacles.

Interestingly, the earliest versions of Erlang lacked most of the features that an academic, semantically-minded functional programmer might consider essential: static typing, higher-order functions, referential transparency and fast compiled code. But Erlang did have many other other features which were crucial for its success: concurrency, distribution, dynamic code loading, interoperability, good tools, training and support.

Other functional language implementations are also maturing into useful tools. For many years, compiler technology was a major focus of functional programming research because poor performance was considered the main obstacle to the wider use of functional languages. This research paid off – modern compilers for functional languages produce code with performance that is usually within a modest factor of that of code compiled from C – but languages like Erlang, Java, Perl and Visual Basic have shown that speed is not always necessary for a new language to be successful, and is certainly not sufficient. The emphasis has now shifted to making functional languages more usable and demonstrating their usefulness. Functional languages now interoperate with other systems using COM [19, 32], CORBA [26] and Java [5]. They have solid, useful libraries [2] and often extend the functional paradigm with concurrency [37, 28, 40, 22].

Functional languages are now being used for more than writing compilers for functional languages. The Fox project at Carnegie-Mellon has produced low-level networking software written in SML [7], the Ensemble project at Cornell has used Objective Caml to implement a toolkit for building distributed applications [25] and Elliott and Hudak have developed a very elegant and powerful language, embedded in Haskell, for programming reactive 3D animations [18], and the same ideas are now being applied to controlling mobile robots [38]. There are many other, though still far too few, examples [46].

In areas like these powerful ideas originating in the theory community have proven applicable to practical problems. This has always been true; when semanticists first tried to model languages with jumps, they invented continuations – a powerful generalisation of jumps. Continuations have since become a language feature, for example in Scheme and SML/NJ. But whatever the language, a programmer who learns about continuations gains a useful new conceptual skeleton or *pattern* for thinking about programs. Nice applications of the shift in perspective that continuations give include Halls's use of continuation passing to restructure CGI-based web scripts [23] and Appel and Tolmach's ‘time travel’ debugger for SML [44].

Semanticists have a strong aesthetic sense which favours simplicity, elegance, generality and provable correctness. This ideal is seldom achieved in applications, so when doing applicable theory there often seems to be a choice between either being ‘pure’ and prescriptive or being ‘impure’ and descriptive.[1] This tension between purity and pragmatism is particularly clear in the evolution of functional languages. The designers and implementers of functional languages have tended also to be experts in semantics and type theory, and have been keen to produce languages which are not only efficient and highly expressive, but also have good theoretical properties, such as type soundness and a rich equational theory. Strict languages like ML are usually thought of as being ‘impure’, allowing traditional imperative programming with mutable state and IO to be mixed freely with functional computation, though at the cost of weakening the equational theory. Lazy functional programmers, by contrast, regard reasoning principles as sacrosanct and have only allowed imperative features into languages like Haskell as research has shown how they may be added without compromising existing equations. This gave rise a very pleasing example of the way in which apparently abstract theory can influence practice: Moggi’s original work on using categorical strong monads to structure denotational semantics [34] quickly became first a programming style [45] and then a language feature [31] and an implementation technique [5]. The tension between purity and pragmatism is a very creative one. From one side, we discover good ways to add impure features to a pure system and from the other we devise new theories to bring impure systems within the scope of our reasoning abilities. Real progress is made when the two approaches converge on a common solution.

Even in the case of ‘pure’ functional languages, however, the true relationship between formal theoretical foundations and real implementations is itself actually quite informal. Sometimes this is ‘just’ a matter of scale: a difficult theorem which establishes the soundness of a new language construct or optimisation is often proved only for a much-simplified λ-calculus, but in such a way that most experts are confident that a correctness proof for the full language *would* work if only one could be bothered with all the details. At other times, the practitioner’s reach really does exceed the semanticist’s grasp. For example, the safe encapsulation of non-interfering state threads in Haskell [29] is based on non-trivial theoretical results about parametricity but, whilst there is no reason to believe it to be unsound, we do not yet quite have the theoretical machinery to establish its formal correctness for the full language including mixed-variance recursive types. Of course, this is perfectly natural situation of the sort which leads to further progress; the point is merely that often what is actually transferred between theory and implementations, even in area where the two are especially close, is ideas rather than theorems.

All of which might prompt our ‘real worlder’ to ask:

> If theory influences practice by the transfer of these mysterious and informal ‘patterns’, then what’s the point of doing all the difficult formal mathematics? Isn’t this a bit like trying to justify the space programme on the basis that it led to non-stick frying pans? Surely

[1] As an unappreciated genius, quoting Shaw, might put it: “The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore, all progress depends upon the unreasonable man.”

we could invent these patterns just by thinking informally about programming?

This is wrong for two reasons. Firstly, computing really is a difficult and unforgiving discipline and there are some things which it is important to get right. Perhaps not all programmers need a deep understanding of the mathematical foundations of programming, but programming language designers, for example, certainly should not just trust to 'common sense', as experience shows that often leads to mistakes (e.g. the unsoundness of the Eiffel type system [11]). The second, and most important, reason is that informal thinking just doesn't scale. One can build high by putting a small, ramshackle structure on top of a tall, solid one, but one cannot build a tall structure which is ramshackle all the way up.

We wouldn't even have the language to *think* about the way encapsulated state is dealt with in Haskell if it weren't for a great deal deal of background theory concerning program equivalences, polymorphism, parametricity, representation independence, logical relations and monads. And that work *had* to be formal because that's the only way we can build such complex towers of ideas which actually work. There are many other examples of clearly applicable ideas which would be inconcievable, or at least unimplementable, without the language and techniques developed by the theory community:

Partial evaluation: Partial evaluation is the process of automatically specializing general purpose programs to take advantage of partial knowledge of their inputs [27, 12]. This is a powerful technique which has been successfully applied to produce compilers from interpreters and to speed up applications ranging from aircraft crew planning to ray tracing and pattern matching. The basic idea comes from classical recursion theory – Kleene's *s-m-n* theorem – and modern partial evaluators also exploit more recent developments in, for example, type theory.

Proof-carrying code: One promising approach to the problems of mobile-code security is to transmit not merely executable code, but also a formal proof that the code will adhere to a particular security or resource usage policy [35, 36]. This proof can typically be checked by the receiver far more efficiently than it could be synthesized. Proof carrying code research depends crucially on many ideas from logic, theorem proving, type theory and programming language semantics.

Region-based memory management: The software engineering benefits of automatic storage management are now widely recognised, but a radical alternative to traditional runtime garbage collection has recently emerged from the theory community. Region-based memory management [43] uses a sophisticated non-standard type inference system to analyse object lifetimes and insert memory allocation and deallocation code statically, at compile time. A version of this system is built into the current version of the ML Kit compiler and shows dramatic performance gains over traditional garbage collection for some programs.

Domain theory and real arithmetic: Classical domain theory has been succesfully applied to the semantics of programming languages since the 1970s. A new kind of domain theory, leading to novel computational techniques in continuous spaces (such as the reals) has recently emerged from a combination of ideas from theoretical computer science with more traditional analysis, topology and measure theory. New algorithms for integration and for infinite-precision real number computation are amongst its results [15].

It should also be emphasized that not only does theory need to be done in a formal way (even if it ends up being less formally applied), but the formal work has its own quite distinct driving forces. Ideas in the formal world are often developed, refined, generalised with little thought being given to applications. The truly marvellous thing is that the new ideas which result do so often prove applicable. In the context of the physical sciences, the mysterious fact that mathematical reasoning actually works for deriving new true facts about the natural world is often referred to as 'the unreasonable effectiveness of mathematics' after a famous paper by Wigner [48, 24]. As information processing systems are human constructions, it is perhaps slightly less mysterious that mathematics is effective for reasoning about them (Milner [33], discussing the development and applicability of semantic idea in computing, characterizes computer science as a 'science of the artificial' [41]). Nevertheless, it is striking how often ideas in computing can be discovered (or sometimes rediscovered) through purely mathematical or logical research. A typical small example is Bierman and de Paiva's term calculus for an intuitionistic version of the modal logic S4 [6], which was derived by categorical and proof-theoretic means – specialising some earlier work on linear logic – and was later discovered to be applicable to staged computation, a close relative of partial evaluation [39].

3 The State of the Art

So far, I have mainly discussed the applicability of logic and semantics research. We should also consider the current state of the art in the wider community. Does industry actually have a need for any of these theory-derived ideas, and, if so, does it know it needs them?

The need is, I think, beyond doubt. The are some small niches, such as programming toasters or maintaining legacy Cobol programs, where the pressure to adopt new programming techniques is small.[2] And there are some sectors, such as microprocessor design, which are already making good use of theory. But for much of the software industry, the revolutions of the last two decades – the introduction of graphical user interfaces, the spread of computers to desktops and homes and the emergence of network computing – have led to chaos (albeit a chaos whose effects are masked by staggering economic growth and an exponential increase in computer power). Instead of the stable towers of precisely interlocking ideas to which I idealistically referred earlier, we have built intellectual shanty towns; quick to construct and displaying much ingenuity, to be sure, but you wouldn't want to work there.

Monolithic, sequential applications running on static and isolated computers are giving way to concurrent, distributed, and even mobile, software components

[2] Actually, even that is untrue: toasters will soon be networked 'smart appliances', and both microcontrollers and legacy Cobol suffer from Y2K problems which, as we've seen, can usefully be addressed by type theory.

running on a dynamic global network of hetrogenous computing devices. The people who once wrote green-screen database applications are now being asked to write 'fault-tolerant, secure, web-enabled agent-based enterprise workflow solutions' and they need all the help they can get. Software is now built using many different languages and technologies all mixed together: C,C++ and now Java are supplemented by complex GUI APIs, DLLs, component frameworks like COM and CORBA, HTML, DHTML, CSS, XML, scripting languages like Perl, Python and Tcl, browser plugin APIs, server extension frameworks, application servers, query languages like SQL and special-purpose specification languages like ASN.1. Ad hoc metaprogramming is common and the universal datatype has become the string.[3] In the 1980s, many programmers started spending more time on GUI programming than on application logic; now they are likely to spend most of their time fighting their middleware.

Vast resources have been devoted to solving the problem of making it easier to produce and maintain increasing amounts of increasingly complex software. There are now graphical development environments that produce user interface code automatically, sophisticated source-level debuggers and analysis tools for tracking down memory leaks and deadlocks. We also had a range of programming fashions, including object-oriented programming, object-oriented design and modelling methodologies (including much which seems to be the IT equivalent of psychobabble), design patterns, component-oriented programming and agent architectures.

It is sometimes suggested that industry is resistant to change, but the speed with which each of these new ideas has been embraced as the new 'silver bullet' suggests otherwise. Unfortunately, non-trivial software is not obviously becoming more reliable or cheaper to produce or maintain as a result. A significant part of the explanation for this is certainly that expectations have increased, but I also believe that few of the proposed solutions even begin to address the fundamental complexity of thinking about computational structure. As Milner [33] puts it, software engineers often erroneously view software as a hydraulic engineer might view water: homogeneous and well-understood, with the main problem being how to deliver it efficiently to the right place.

The unsatisfactory state of software development is *not* a consequence of widespread stupidity in the industry. On the contrary, the fact that software systems of hundreds of thousands of lines of C can be made to work at all is a marvellous demonstration of human intelligence and ingenuity. Furthermore, there are usually perfectly reasonable explanations for things being done the way they are, including a shortage of trained staff, the need for well-supported tools and the fact that computer systems are typically developed under great time pressure, so that the local optimum is always to stick with concepts and technologies with which one is already familiar.

One interesting recent development, which shows industrial recognition of the need for powerful and, above all, *reusable* ideas is the rise of the 'design patterns' movement [21]. Design patterns are attempts to identify and classify useful programming structures at a slightly higher level than code in a particular programming language. A theoretician who looks at patterns like 'command', 'action', 'iterator', 'visitor' or 'interpreter' will see clumsy expressions of familiar

[3] No language theorist who has seen Embedded SQL could doubt that there is useful work to be done, and I have seen complex datastructures made persistent by shoehorning them into relational database tables: if all you have is a hammer, everything looks like a nail.

ideas such as higher-order functions, continuations and homomorphisms. Here, theorists seem to have produced the ideas which practitioners want, but to have failed to communicate them.

4 Ways Forward

So how can theory go about influencing practice more effectively? There is a competitive marketplace for ideas in computing and I believe the theory community benefits from devoting a certain amount of its energy to capturing what marketing people call 'mindshare'.

As well as doing big, impressive things like formal verification of safety-critical systems, it is important to popularize all the little ideas which are taken for granted in the theory community. These include continuations, invariants, higher-order functions, thinking with types, languages for concurrency, the very idea of program equivalences, equational reasoning, polymorphism, initial algebras and induction. These are, of course, largely things which undergraduates are supposed to learn in university computer science courses, but my own experience is that even good students often fail to see a connection between these apparently esoteric subjects and the everyday practice of programming. So a better integration of theory courses within university curricula is desirable. To achieve that, and to reach a wider audience of practising professionals, theorists need to spread their ideas by example.

Useful examples do not have to be explicitly presented as applications of theory. Simply solving a practical problem in a 'theory-aware' style is often more effective. A case in point is Elliott's work, referred to above, on programming reactive animations in Haskell. Elliott's approach is based on a great deal of knowledge of type theory and functional programming, but is immediately appealing to programmers who know little of either. Another aspect of Elliott's work which makes it an exemplar of how good ideas should be spread is that it has also been written up in a publication which is actually read by practising programmers [17], rather than being confined to the academic literature.

Useful examples also do not have to offer a perfect solution to every aspect of a problem. The scientific culture of rigorous review and academic honesty[4] can make researchers reluctant to proclaim their ideas as perfect solutions when they know perfectly well that this is not the case. Unfortunately, these ideas compete with weaker ones that are hyped with no such reservations. I am certainly not advocating dishonesty and hype in research, merely observing that ideas that might seem slightly inelegant or that are imperfectly understood can still be usefully applied.

There are many areas in which theoreticians can find ways to apply their work, and recent industrial developments have created plenty of exciting new opportunities. For example:

Security Logic and semantics researchers have always been concerned with correctness but have found it hard to convince most of the rest of the world that producing programs which provably meet their specifications is either possible or worth the effort. The spread of local and global networks has, however, made security a major concern and this seems to

[4] A somewhat idealized view, of course...

be an area in which formal techniques can gain widespread acceptance. I have already mentioned the Java verifier, ambients and proof-carrying code and there are many other examples of new and existing theory being applied to security problems. There is undoubtedly plenty of scope for further applied work in this field.

Concurrency, Distribution and Mobility These three related ideas are not new, but have gained a new importance. A few years ago, concurrent and parallel programming was fairly esoteric, distributed applications were mainly written by systems researchers and both mobile code and mobile computers were practically unknown. Now they are all widespread. Such systems are far harder to design, implement and test than traditional single-threaded applications, which offers an opportunity for new programming paradigms to gain acceptance. Theoreticians have been studying concurrency for many years and have developed a significant body of knowledge and a number of tools (e.g. model checkers) which have been successfully applied to real applications, such as verifying network protocols. Distribution and mobility have received similar attention more recently.

From an intellectual point of view, concurrency has always been one of the main strands of theoretical research and, along with ideas from other areas of logic, semantics and type theory, it is a large part of what Milner, Abramsky and others have identified as an emerging 'science of interaction' [33]. Where this will lead in the long term is impossible to say, but what is certain is that in the short term more ad hoc approaches to the same essential problems will proliferate. Continuing to develop and apply theory-based tools and languages for concurrent, distributed and mobile applications is thus useful today and helps ensure that solutions developed tomorrow have a chance of being recognised as such.

Domain-specific languages Although large applications are mainly written in a relatively small number of languages, there is an ever-growing Babel of 'little languages' for more specific tasks: music, typesetting, graphics, web scripting, defining component interfaces, filtering network packets, querying databases, describing hardware and so on. These can provide an excellent vehicle for testing and spreading new ideas about programming, language design and optimization because they are simpler to implement and far easier to gain acceptance for than new general-purpose languages.

This is emphatically *not* a suggestion that theoretical research should be largely directed towards the solution of immediate engineering problems, or that semanticists should all give up proving theorems for a few years in favour of writing applications. Even if one is only interested in industrial efficiency, I hope I have indicated how truly basic theoretical research can make a significant contribution. That potential will remain largely unrealised whilst theoreticians and practitioners speak apparently different languages. The right way to bridge the gap is for theoreticians to spend *some* time demonstrating the utility of their approach to computer science and I hope I have also shown how many of them are succesfully doing just that.

5 Obstacles

The preceeding discussion could fairly be characterized as 'motherhood and apple pie' – nobody would really argue that basic scientific research with demonstrable application to practical problems is anything other than A Good Thing. But succesfully transferring theoretical ideas into more applied research is certainly not easy, and getting them into industry is very hard indeed. Amongst the obstacles are:

1. Nobody got fired for choosing C++. Many computing projects fail, and if everything is done in a conventional way then this is regarded as normal. If, however, some new technology has been employed and the project fails, then that technology will be blamed and the person who decided to adopt it punished.

2. It's got to be done by next week. Commercial software is typically developed to very tight deadlines, so there is insufficient time for a contemplative theoretician to break into the development cycle, analyse a problem and offer a good solution – the first thing that seemed to work will be set in stone by then. One way around this is to have the answer ready before the question is asked.

3. Suspicion of 'clever-clever' solutions. By and large, industry is interested in solutions to immediate problems, not in beautiful ideas. So it's important to present things in the right way. ("Well, I've designed this new programming language," is, unfortunately, nearly always a bad way to start.) It is also dangerous for an idea to look too good to be true, too unconventional or frighteningly sophisticated.

4. Immature tools, support and documentation. Software that is produced by researchers as a proof-of-concept is, unsurprisingly, usually unsuitable for use in real applications. It is often buggy, has a poor user-interface, fails to interoperate with other systems, lacks essential features and is unsupported and badly documented. The fact that it is probably free does not make up for these shortcomings. The way around this problem is either to try to convince a company to develop and support a commercial version, or for the research group to undertake a more serious long-term development and support role. There are successful examples of both approaches, but the latter carries a particular risk:

5. Damage to a research career. Developing and supporting a significant piece of software is not seen as 'real research', so those who choose to do so run the risk of damaging their publication record. Peyton Jones [30] argues (and indeed demonstrates) that the long-term development of serious research platforms, rather than throwaway prototypes, can lead to important new research being done, and should be more widely encouraged.

6. How will that help my CV? Individual programmers moving from job to job like to have CVs filled with widely-recognized standard skills, so can be reluctant to use new techniques even when they believe them to be useful.

7. Manager power. Decisions about what technology should be used on a commercial project are taken by managers (who often don't have to use it themselves). The perceived power of a manager is measured by, amongst other things, the size of the team working under her. Hence she has little incentive to adopt technology which will enable the job to be done with a smaller team.[5]

8. It is not enough to succeed, others must fail. One of the lessons learnt by the Erlang team is that it is often not sufficient to propose a demonstrably good solution to a problem using new ideas. Natural conservatism means that a more traditional approach will still be chosen. Instead, one has to wait for the traditional approach to actually fail, and then 'save the day' at the last moment.

Some of these problems are illustrated by a personal anecdote. Audrey Tan and I were once asked to design a visual tool for writing a certain kind of query on event streams from a web server. We did everything I have advocated: drawing on ideas from dataflow, proof nets and functional programming (but keeping that quiet), we designed and implemented a simple and elegant graphical query language based on a small family of list processing cominators. The system was written in SML and Java, had a neat graphical user interface and did the job beautifully. At first we were asked a lot of questions about how particular queries could be expressed in our system, and we dealt with all those. Then the question of performance was raised, so we demonstrated that it ran quite fast enough and that we could make it go considerably faster in future if that ever became necessary. Next it was suggested that it wouldn't integrate with the rest of the system, so we built a live demo linking our prototype into a real web server. Then doubts were expressed about the use of SML, particularly regarding maintainability. This was much harder to deal with, even though the company had a number of programmers who knew SML. But there was no reason, apart from taste, why the prototype couldn't be reimplemented entirely in Java. Ultimately, none of this was enough to overcome the fear that our solution was overengineered, too unusual and came with too much mysterious complexity hidden somewhere beneath its friendly interface. Our approach was dropped in favour of a simple set of hardwired queries with which the developers felt more comfortable.

I do not have neat ways around all these obstacles. Successful pattern transfer takes good ideas, perseverance and a good deal of luck. But each success makes the next one easier to achieve.

6 Conclusions

Basis research in theoretical computer science can make a great contribution to computing practice. But that will only happen if theory ceases to be viewed as something abstruse and irrelevant and is instead recognized as the natural result of thinking about computation using a methodology which has proved itself spectacularly effective across engineering and the physical sciences. Experience shows that even quite simple bits of theory can be usefully applied, and this can

[5] I owe this lovely example, like several of the others, to a great talk by Joe Armstrong of Ericsson.

be just as effective in terms of gaining wider acceptance of theoretical ideas as more ambitious formal projects.

I have said very little about concurrency, theorem proving or hardware design, despite these having some of the best examples of a healthy interaction between theory and practice. This is simply because I don't know nearly enough about them. I have also not discussed the use of semantic ideas in optimizing compilation, although that is one of my interests and another major way in which theoretical results can make a direct contribution to practice. However, in the context of popularizing theory, the internals of compilers for functional languages probably count as a bit too introspective.

Despite all the potential obstacles, doing some application work can also be highly enjoyable and lead to new perspectives on research issues.[6] And theoreticians should not worry too much about spending some time on things which don't seem to be 'proper' research – any successful application work will be seized on with delight by the rest of the theory community as further justification for their own work.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages and Applications*, 1997.

[2] A. W. Appel, N. Barnes, D. Berry, E. R. Gansner, L. George, L. Huelsbergen, D. MacQueen, B. Monahan, C. Muller, J. H. Reppy, P. Sestoft, and J. Thackray. The Standard ML basis library. `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html`, 1997.

[3] J. Armstrong. The development of Erlang. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, June 1997.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[5] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN Conference on Functional Programming*, September 1998.

[6] G. M. Bierman and V. C. V. de Pavia. Intuitionistic necessity revisited. In *Proceedings of the Logic at Work Conference, Amsterdam, Holland*, December 1992.

[7] E. Bigioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a network protocol stack: A systems application of standard ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1994.

[6]For example, I personally became much more interested in metaprogramming as a result of thinking about a stylesheet language for active documents.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, October 1998.

[9] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Conference Record of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, January 1999.

[11] W. R. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4), 1989.

[12] O. Danvy, R. Gluck, and P. Thiemann. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[13] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy*, May 1996.

[14] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 1998.

[15] A. Edalat. Domains for computation in mathematics, physics and exact real arithmetic. *Bulletin of Symbolic Logic*, 3(4):401–452, 1997.

[16] P. Eidorf, F. Henglein, C. Mossin, H. Niss, M. Sorensen, and M. Tofte. Anno Domini: From type theory to year 2000 conversion tool. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, January 1999.

[17] C. Elliott. Composing reactive animations. *Dr Dobb's Journal*, July 1998.

[18] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, June 1997.

[19] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[20] S. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Symposium on Object Oriented Programming: Systems, Languages and Applications*, 1998.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[23] D. A. Halls. *Applying Mobile Code to Distributed Systems.* PhD thesis, Computer Laboratory, University of Cambridge, June 1997.

[24] R. W. Hamming. The unreasonable effectiveness of mathematics. *The American Mathematical Monthly*, 87(2), February 1980.

[25] M. Hayden and R. van Renesse. Optimizing layered communication protocols. In *Proceedings of the 1997 Symposium on High Performance Distributed Computing, Portland, Oregon*, August 1997.

[26] D. Jeffery, T. Dowd, and Z. Somogyi. MCORBA: A CORBA binding for Mercury. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, San Antonio, Texas*, volume 1551 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1999.

[27] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, June 1993.

[28] S. L. Peyton Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, January 1996.

[29] S. L. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[30] S. Peyton Jones. On the importance of being the right size: The challenge of conducting realistic experiments. In I. Wand and R. Milner, editors, *Computing Tomorrow: The Future of Research in Computer Science*, chapter 16. Cambridge University Press, August 1996.

[31] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org/onlinereport/`, February 1999.

[32] X. Leroy. Camlidl user's manual version 1.0. `http://caml.inria.fr/camlidl/htmlman/`, March 1999.

[33] R. Milner. Semantic ideas in computing. In I. Wand and R. Milner, editors, *Computing Tomorrow: The Future of Research in Computer Science*, chapter 13. Cambridge University Press, August 1996.

[34] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[35] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[36] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems, Design and Implementation, Seattle, Washington*, October 1996.

[37] F. Nielson, editor. *ML With Concurrency: Design, Analysis, Implementation, and Application.* Monographs in Computer Science. Springer-Verlag, 1996.

[38] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, San Antonio, Texas*, volume 1551 of *Lecture Notes in Computer Science.* Springer-Verlag, January 1999.

[39] F. Pfenning and R. Davies. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, January 1996.

[40] J. H. Reppy. Concurrent ML: Design, application and semantics. In *Proceedings of Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer-Verlag, 1993.

[41] H. Simon. *The Sciences of the Artificial.* MIT Press, Cambridge, MA, 1969.

[42] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[43] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[44] A. Tolmach and A. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

[45] P. Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences.* Springer-Verlag, August 1992.

[46] P. Wadler. An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, February 1998.

[47] P. Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, August 1998.

[48] E. P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. *Communications on Pure and Applied Mathematics*, (13), February 1960.