

## **E1 PROJECT TITLE**

“Development of a Dataless, Totally Functional Programming Style”

## **E2 PROJECT DESCRIPTION AND BACKGROUND**

A new motivation for higher-order Functional Programming (FP) [1] indicates new directions for the exploitation of FP, but which requires significant further investigation in order to determine the extent of, and then to realize, its potential.

### **A New Motivation for FP**

The origins of this project lie in the vision of projecting the idea of language extensibility into mainstream software development. It's demonstrable that programming is a language design/extension activity [2]: pragmatically, standard criteria for program quality assessment parallel those for assessing language designs; formally, a straightforward reordering of the parameters to the denotational semantic meaning function exposes declarations as explicit language-extending constructs.

Consequently, it stands to reason that bad language design/extension practices should be avoided in programming. Typifying adversity in language extension is interpretation, whereby new semantics are provided through extension of the syntax of the original base or “host” language followed by extension of the interpreter for the host to include in its domain the extensions of the “guest” language. By comparison, far simpler and thus preferable is direct definition, whereby the association of new syntax with new semantics is provided by a local replacement rule, such as macro expansion or identifier declaration that binds the name of the new semantic entity to a term that provides the semantics as directly expressed in the host. In order that this “definitional” approach permits universality (with respect to some desired semantic domain), it's necessary that the host language be “expressively complete” [3]. For Turing-computable functions, a necessary condition for expressive completeness is the capacity to define arbitrary higher-order functions, which reflects the fact that the typical practical demand for interpretational extension arises from the need to express processes or operations (modeled as higher-order functions) but which are inexpressible in usual first-order (i.e. non-FP) host languages. In other words, FP is justified by its enablement of definitional as opposed to interpretational language extension.

### **TFP - a New Direction for FP**

From the hypothesis that programming is in fact a kind of language extension it's further to be expected that the distinctions between interpretational and definitional language extension would have images in the distinctions between interpretational and definitional programming, and that FP should be the key enabler of the last of these as well. So, just as definitional extension eschews the interpretation of data (that would be needed to implement arbitrary functions), so would definitional programming eschew data, replacing them by functions. In other words, definitional-style programming (the image of definitional language extension) would appear to be a kind of FP, i.e. “Totally Functional Programming” (TFP) in which *total* reliance is placed upon (higher-order) functions, as opposed to a mixture of (higher-order) functions and symbolic data, as is the case with “normal” (i.e., interpretational) FP.

The hypothesis implicit in this vision is that just as the data to which an interpreter applies denotes some applicative behaviour which is animated by the interpreter, so does the data in a program represent some behaviour which the program animates. The essential advantage of definitional-style TFP would be that programs (and programmers) would not have to animate behaviours from data, rather the pre-animated behaviours inherent in functional representations (which we call “platonic combinators”) could be directly applied. (Observe how the representation of data by functions necessitates a higher-order functional capability for the representation of operations on the “data”.)

There is in fact substantial evidence (see below) in support of this hypothesis, that many “data structures” inherently represent processes that could and should be directly represented as functions. Needless to say an important part of the proposed research is to explore the extent of the validity of this hypothesis.

Of course, the complete abolition of symbolic data would appear to be an unreasonable goal. Thus, the technical developments will always emphasise mixed development: interfacing between totally functional components (“pure platonic combinators”) and those in which data and functional representations co-exist, and indeed hybrids in which TFP representations may retain attributes pertaining to symbolic data (“impure platonic combinators”)

### Technical Questions for TFP

A constructive substantiation of our hypothesis involves the development of totally functional programs as well as practical development methods and implementation techniques. The ten key questions around which our research is structured are as follows.

1. *How much of FP is actually TFP?* One of the abovementioned interesting examples of advanced functional programming that apparently involves the replacement of symbolic data by functions is Boehm and Cartwright’s exact real arithmetic [4]. Perhaps even more obviously related to our goal of replacing data-interpretation combinations with direct function definitions are the “combinator parsers” of Hutton [5] and others, in which the usual combination of a grammar (or equivalent representation such as a parse table) and its interpreting “parsing engine” are replaced by a collection of parsers, one for each nonterminal symbol in the grammar. Context-free operations of grammar concatenation and alternation are implemented by appropriate higher-order functions on parsers. We anticipate that understanding of the extent to which TFP can be discerned in other advanced examples of FP will illuminate solutions to several of the other questions posed here.
2. *What is the precise distinction between Definition and Interpretation?* In general, the universality of data-less functional languages (such as the lambda-calculus) means that simply abolishing data is insufficient to prevent the writing of interpreters, and in particular there are simple lambda-terms that appear to treat functions as symbolic data (e.g. testing for specific behaviours, such as zero-testing of Church numerals). What is a precise, even syntactic characterization of “interpretation”, at least so that we know certainly what TFP is, and so that a subset of FP languages can be identified that will be adequate to support TFP? NB by “defining out” the capacity to write interpreters, it would seem obvious that such a subset would be subrecursive.
3. *How may functional alternatives for data be derived systematically?* Corresponding to the substantial and long-standing literature about the derivation of implementations (= representations + operations) from abstract data type specifications, there also needs to be a systematic way of deriving the TFP combinations of representation and implementations, in particular the variants of the ADT constructors that generate platonic combinators rather than data structures.
4. *What is the semantic model for TFP?* An elegant semantics for any FP subset for TFP would serve to validate the characterization of interpretation vs definition, as well as providing a good technical foundation for any specialized approach to verification of TFPs.
5. *How may TFP be type-checked?* Experience with even some elementary TFP examples (arithmetic operations on natural numbers) indicates that more powerful type-checking than usual in FP is required.
6. *How may functional language implementations be optimized for TFP?* If TFP can indeed be identified with a functional language subset, then it’s possible that the full generality of functional language implementation infrastructure may not be required.

7. *What additional primitives are necessary?* It emerges that the TFP language subset will almost certainly be subrecursive (see above), in which case it may be useful to select carefully additional useful primitive operators to ensure applicability of our subrecursive functional language in popular domains
8. *How may functional language verification and validation be optimized for TFP?* Just as dealing with an FP subset may allow implementation optimization, so may subsetting permit simpler proofs, which we would expect to be founded in the simpler semantic model that is also sought (4. above).
9. *Is TFP more widely-applicable?* By basing computation on the applicative/dynamic properties of abstractions, rather than on the interpretation of symbols, TFP may have wider applicability than just computer programming, i.e. in other domains where the composition of systems is defined in terms of the behaviours of their components (see “Significance Beyond Computer Programming” below).
10. *Is TFP compatible with the rest of software engineering?* We’ve already emphasized that the two styles, “data-less”/definitional TFP and “data-full”/interpretational programming need to co-exist. More generally, programming methodologies that are in their essentials independent of computation model (e.g. object-oriented design should be accessible from procedural, logic, functional, etc. programming) should equally be accessible from within TFP.

The remainder of this application is structured around these questions and their envisaged solutions or means of discovering solutions.

## **Related Work**

Our project unifies three related but separable streams of earlier work:

1. a strong tradition of “subrecursive” programming [6], in which all programs are mathematically “total” (i.e. terminating) functions, and which was later in effect recast in a pure functional context [7];
2. higher-order typed lambda-calculus [8], in which the parametric polymorphic type system is so expressive that all conceivable arithmetic functions can be defined without recursion;
3. packaging of recursion over recursive data types (e.g. naturals, lists, trees) in terms of the “fold” functions on those types [9].

Our research is most directly a development of 2. The link between 1. and 2. is obvious, but the subtle link between 2. and 3. is key: (partial) application of a fold to the elements of a data structure but with the other operands left unspecified that define the specific computation, is the same functional representation of a data structure that is used in higher-order typed lambda-calculus.

Our research vision can be thought of as taking the above theoretical basis and demonstrating its practical applicability as follows:

- extension of functional representations for data beyond those equivalent to “folds” only;
- improvement of practicality of higher-order typing (e.g. raising bounds on what types can be inferred as opposed to being required to be provided explicitly by the programmer);
- taking advantage of any other implementation advantages from dealing with a functional language subset (e.g. no recursion);
- extension to other domains (design recovery, analog design, systems engineering)

together with appropriate consolidation of theoretical foundations.

## E3 SIGNIFICANCE AND INNOVATION

### Problem Significance

**Programming is significant.** There can be little doubt that “programming” (both as metaphor for and essence of the entirety of software development activity), after a half-century of unceasing research, remains too complex. The plethora of complaints about - performance of software systems; inefficiency of software development; and the proposed remedies that have failed to accomplish dramatic change - all encourage the search for a new view that will yield significant improvement.

The goal of this project accordingly is to discover a radical simplification of software development. Our approach is to achieve this by exploring the functional paradigm to its limits, distinctively in our case through replacement to the greatest extent possible of non-functional components, i.e. data and data structures, by appropriate applicative alternatives. This will be done by discovering functions that implement the essential applicative behaviour that is hypothesised to exist when processing each data (structure) type. The choice of functional programming as a basis suggests itself in formal terms because it inherently supplies most *convenient* access to the richest range of computable functions compared to other paradigms (see discussion of “expressive completeness” below). Further, functional programming is pragmatically suggested by the paradigm’s demonstrable promise as a vehicle for simple, verifiable solutions to complex programming problems.

**Functional Programming is Significant.** “Functional” programming [1] is about the exploitation of function definition and application in software development. It traditionally therefore includes “applicative” programming, but extends that paradigm to programmer-definable higher-order functions, i.e. functions with functions as their arguments and distinctively with functions as their results. All of the apparent characteristics of applicative/functional programming (e.g. recursion, lazy evaluation) can indeed be explained as particular cases of higher-order functions. Use of higher-order functions comprehensively (or “totally”, as in the proposal title) to replace data structures is logical fulfillment of this scheme of things. Our extensive ambitions for functional programming go beyond “academic” interest, and in that are well-founded as shown by how the functional paradigm has repeatedly confounded critics of its “real world” utility [10, 11]. Key developments pertaining to the practicability of functional programming for software applications development include: applicability of functional programming to “real” problems as diverse as parsing [5] and multimedia [12]; efficient implementations via (super-) combinator compilation [13]; occupation of a distinct niche in the software life cycle [14]; practical interactive I-O via “monads” [15]

As well as the general advantage that can be expected to flow from replacing the current separate bases of programming languages (functions/programs + data viz. Griswold’s linguistic schism” [16]) with a single uniform functional basis, specific consequent advantages can also be expected. In our case, we promise (as will be detailed below): (a) simpler programs; (b) possibly more efficiently-implemented and easily-verifiable programs; (c) extended application of functional programming beyond software engineering. There are many facets of programming that could be, and indeed have been addressed as targets for revolutionary overturning of traditional practice, e.g. logic programming addresses the directed nature of the input-put relations, concurrent programming addresses the limits of purely sequential views of software design, object-oriented programming aims for better modeling of domain ontologies. We admit that no single technological development has yet resulted in a global reduction in the complexity inherent in programming. However, focussing on single dimensions of technology in each of the above cases has yielded productive insights. Hence, our similar focus on extending the use of higher-order functions is a legitimate research paradigm-cum-goal. Moreover, the fundamentality to programming of the issue we are addressing (i.e. questioning the very need for data at all) if anything amplifies the likely significance of the results we seek.

**Significance Beyond Computer Programming.** If our essential idea, of programming entirely by assembling applicative behaviours of subcomponents, is valid, then other systems so formed should be able to be specified using the same methodologies. This will be tested by application in analog and

systems domains. Indeed, there is a growing awareness of the applicability of computer science/software engineering methodologies in other branches of engineering, viz. recent international workshops on *Integration of Specification Techniques for Applications in Engineering* [17] and *Semantic Foundations of Engineering Design Languages* [18].

### Novelty/Innovation of Outcomes, Aims and Concepts

The essential innovation we propose is simple but far-reaching: to replace the hitherto-prevailing “Turing” view of computation, whereby computational behaviour is routinely derived from the interpretation of symbolic data. It’s thus illustrative to review some of the details of how TFP/platonic combinators work. Consider for example, when a natural number  $n$  is represented as a Church numeral  $C_n$ , i.e. as an  $n$ -fold functional composition operator. Thus: “ $C_n f x = f (... n \text{ times } ... (f x)...)$ ”. Accordingly, individual Church numerals are created by generator functions corresponding to the constructors for the otherwise standard “natural” type, i.e., corresponding to

$$\text{type Nat} = \text{Zero} \mid \text{Succ Zero}$$

we have definitions

$$\begin{aligned} 0 f x &= f x \\ \text{succ } n f x &= f (n f x) \end{aligned}$$

The import of this functional representation is that it embodies the view a natural number *is* an iterator, and that every usage of a natural number *qua* natural number is to iterate. This is where our term “platonic combinator” arises for such representations: Church numerals embody the essence of what it is to be a natural number, that is, an iterator. (It might contrarily be argued that natural numbers perform other useful roles in computing, e.g. as symbols identifying different values, but one of the first things students learn in programming is to use enumerated types for such applications!)

Platonic combinators simplify programming as follows. In conventional interpretational programming, in order to compute with naturals, it’s necessary both to interpret iteration from the symbolic representation, and then to perform the calculation required by the application/domain context. By comparison, with definitional TFP, the required iterative behaviour results from application of the functional representation of naturals as Church numerals. For example, consider the interpretational implementations of arithmetic operations (assuming only primitives of zero and successor/predecessor):

$$\begin{aligned} \text{add } n1 \ n2 &= \text{if } n1=0 \text{ then } n2 \text{ else succ (add (pred } n1) \ n2) \\ \text{mul } n1 \ n2 &= \text{if } n1=0 \text{ then } 0 \text{ else add (mul (pred } n1) \ n2) \ n2 \\ \text{pow } n1 \ n2 &= \text{if } n2=0 \text{ then } 1 \text{ else mul (pow } n1 \ (\text{pred } n2)) \ n1 \end{aligned}$$

vs their definitions TFP-style:

$$\begin{aligned} \text{add } n1 \ n2 &= n1 \ \text{succ } n2 \\ \text{mult } n1 \ n2 &= n1 \ (\text{add } n2) \ 0 \\ \text{exp } n1 \ n2 &= n2 \ (\text{mult } n1) \ (\text{succ } 0) \end{aligned}$$

In the interpretational implementations of “add”, “mul” and “pow”, not only is there more obvious complexity, but even the structure of an interpreter (recursion and branching over the representation) is evident.

What makes TFP more than just a theoretical curiosity is firstly that platonic combinators and their generators can in fact be derived, not just for natural numbers, but for all regular recursive types, and indeed are closely-related to the appropriate version of the familiar list “fold” function that exists for all such types [9]. Indeed, the advantage of programming with “folds” can be thought of as an explicit

disentangling of interpretation from context, whereas the advantage beyond “folds” of TFP is that we remove the need for interpretation itself. Secondly, we observe the extension of platonic combinators derived from pure structures as above, to include “impure” platonic combinators that account for extractor/inspector/selector operations on structures. A good example of this is the “set” abstraction, with membership test inspector, for which the impure platonic combinators are the characteristic predicates of the sets so defined, and for which the generators are:

empty  $x = \text{False}$   
singleton  $e\ x = e=x$   
union  $s1\ s2\ x = (x\ s1)\ \text{or}\ (x\ s2)$

Note that there is no longer any need for a distinct “member” operator, as sets are directly applicable to putative members. Note also that there is a complementary sense in which characteristic predicates are “impure” (platonic combinators), in that membership testing involves symbolic comparison of actual vs putative members (see body of “singleton”).

Significantly, impure platonic combinators appear in a variety of serious application and systems contexts (e.g. context-free parsers, exact real arithmetic), which encourages us in the search for many more, and indeed systematic means for their discovery, as the essence of the TFP programming style.

### **New Methodologies or Technologies**

The outcomes of this project are the new software development methodologies and technologies summarized thus in terms of the above “ten questions”:

1. *How much of FP is actually TFP?* Identification and analysis of advanced functional programming applications aims to explain numerous advanced functional programming techniques in terms of TFP, the idea being that powerful examples of higher-order function usage can actually be perceived as generation and manipulation of platonic combinators.
2. *What is the precise distinction between Definition and Interpretation?* A syntactic characterization of definition vs. interpretation will implicitly define a conformant (probably subrecursive) functional language subset, which can then be used as a basis for further technological and methodological developments, particularly with respect to Q.3-8.
3. *How may functional alternatives for data be derived systematically?* Pure platonic combinators follow automatically from type signatures, but the impure variants are not so obvious. The considerable literature on functional program transformation and derivation should be capable of being adapted for this purpose.
4. *What is the semantic model for TFP?* In the long run, a relatively simpler semantic model may facilitate the development of more automated derivation/verification/transformation tools, as well as dictating the expressiveness of the FP subset for TFP.
5. *How may TFP be type-checked?* Reynolds’ second-order polymorphic types [8] admit a much-wider class of functions than standard Hindley-Milner type checking [19], but suffer the practical drawback of requiring explicit nomination of types by programmers, rather than allowing the convenience of type inference. What advances in type inference are mature enough for employment here? It’s known that second-order polymorphic type inference is ultimately undecidable, but that for important subclasses (e.g. “rank 2”) this is not the case [20]. Even if “rank 2” is suitable for TFP, a better implementation than is currently available [21] is indicated, that avoids unnecessary encoding of functions as data structures!
6. *How may functional language implementations be optimized for TFP?* In other words, more efficient implementations may be possible by exploiting the functional language subset.

7. *What additional primitives are necessary?* It's well-established that a usefully-large class of arithmetic is available for TFP (e.g. from 2<sup>nd</sup>-order polymorphic typed lambda-calculus) and presumably counterparts for other simple types (e.g. lists), but we may have to invest some effort in discovering additional appropriate operations (e.g. parallel operators?) to achieve expressive completeness with respect to the semantic domain (4. above)
8. *How may functional language verification and validation be optimized for TFP?* See 4. above.
9. *Is TFP more widely-applicable?* Briefly, yes! A programming style based on composing compents characterized entirely by their (applicative) behaviours holds promise as a basis for modeling analog electronic design and broader systems engineering. See "Significance Beyond Computer Programming" above.
10. *Is TFP compatible with the rest of software engineering?* While the admission of impure alongside pure platonic combinators would at least allow definitional TFP to be introduced gradually alongside conventional definitional programming, there is at least an apparent unresolved conflict, with object-oriented programming. Platonic combinators correspond to objects upon which a single method is defined – the applicative behaviour of the object, i.e., the behaviour inherent to the data that the platonic combinatory embodies, *is* the method. Consider for example how the "member" method on sets was subsumed into the characteristic predicate representation. In contrast, OOP permits numerous methods to be defined on objects. We aim for a presentation of platonic combinator derivation that relates to a notion of the cohesiveness of methods for an object/class in order to justify single-method objects/classes.

## **E4 APPROACH**

This project will be conducted according to stages as follows.

### **Foundations**

The first stage of the project proper will be to resolve the foundational questions, following anticipated directions as follows.

1. *How much of FP is actually TFP?* These results will be key inputs to 2. and 3. below ....
2. *What is the precise distinction between Definition and Interpretation?* Our current expectation is that we will be able to identify an (illegitimate) functional simulation of data as being one in which a function leaves no residual in the result of its application.
3. *How may functional alternatives for data be derived systematically?* With one of our PhD students, we've already exploited the affinity between platonic combinators and "fold" functions in order to show how impure platonic combinators can be derived [22] using "fusion" [9]. What are the limits of the "fusion" technique for impure platonic combinator derivation, and what techniques are necessary otherwise.

### **Infrastructure**

In these stages, the promise of Total Functional Programming to software engineering will be demonstrated through the development of enabling technology and significant new applications.

4. *What is the semantic model for TFP?* There are numerous domain constructions alternate to the usual "Scott" domain and its derivatives. Which to develop will depend upon the outcome of 2.
5. *How may TFP be type-checked?* The suitability to TFP/platonic combinators of "rank 2" 2<sup>nd</sup>-order polymorphic types needs to be assessed, and a more integrated implementation produced.

6. *How may functional language implementations be optimized for TFP?* It's impossible at this stage to speculate further, beyond promising optimisations for built-in operations and additional primitives. Further specifics will have to await the outcome of 1. and 2. at least.
7. *What additional primitives are necessary?* For example, just as expressive completeness with respect to the Scott domain induces a need to introduce parallel logical connectives to functional languages [3], what may be necessary for the domain for TFP (see 4. above).

## Extension

Finally, opportunities/needs for enhancement of supporting functional programming technology will be identified and exploited/addressed to the maximum extent possible.

8. *How may functional language verification and validation be optimized for TFP?* Depends upon 3. and 4.
9. *Is TFP more widely-applicable?* Some avenues we propose to explore include: analog design (in which the composition of components based on their physical characteristics enjoys at least superficial similarity to the basis of TFP) adapting our sublanguage and methodologies for that domain; and software design recovery (in which interpretation of data can obfuscate the actual function of a software system), adapting software design recovery tool to generate platonic combinators, and extending its functionality in order to recognize and remove interpretation of data from source programs.
10. *Is TFP compatible with the rest of software engineering?* Likely complementary avenues for reconciling TFP and OOP include such as: from the TFP point of view, discovery of higher-order platonic combinators from which different methods permitted by OOP can be instantiated; from the OOP point of view, a better understanding of the limits on the diversity that be accommodated within an object's methods for the sake of cohesion, and alignment of that understanding with the derivation of the higher-order platonic combinators we seek for TFP.

## E5 NATIONAL BENEFIT

### Outcomes

At the very least, this project will extend the scope of functional programming. And provide a new basis for explaining some of the more powerful functional applications already known. An expected result of the new basis would be the discovery of new elements in the space characterized by the basis, i.e. new functional programming applications. Importantly, these applications would extend beyond software engineering to analog and systems engineering design.

### Impact

The impact of the above minimal goals would make significant contributions, not just to software technology but also to the unification of software engineering with electronic and systems engineering that has been proceeding apace recently. For example, the Software Engineering Institute "Capability Maturity Model" for software development organisations [23] has been embraced by the systems engineering community.

At the most, we promise to demonstrate the viability of a new view of computation that quite overturns the conventional wisdom that has applied at least since Turing [24]. What can fairly be called the "Turing view" is that computation is determined by a fixed machine that processes symbols. However, if the machine is sufficiently powerful (a "universal machine"), these symbols can serve to define another (universal) machine, and the act of processing these symbols in fact effects the operation of this second machine. There would however appear to be two flaws with the Turing view:

1. the multi-stage specification of new computation processes (by inert symbols that need to be interpreted/animated by a lower-level processor, in turn specified etc...)
2. is there any need for a processor of truly universal power other than to implement an interpreter for a another universal machine?

Our project aims to test these flaws (and thereby hopes to overturn the Turing view) by exhibiting a counter-exemplary programming style, where interpretation of inert symbols is avoided and where the nontermination problem with universal machines is also avoided. Ultimately, we aim to realise simultaneously the goals of successive ACM Turing Award lecturers, to facilitate the incarnation of new programming paradigms [25] in the simple context of functional programming [26].

### **Benefit to Australia**

Economic benefits are quite delicately related to individual intellectual advances. Australia has a notorious record for capitalizing upon all sorts of indigenous scientific advance (this applicant has already seen some of his research productised, but offshore), and it would be foolish to think this project would be immune from these risks. On the other hand, there are signal examples of how smaller nations' reputations have been considerably enhanced by specific technical developments (e.g. Finland's mobile-phone-based reputation in high-tech has been considerably reinforced through domestic development of the Linux operating system). Should we be successful in initiating the overturning of the Turing view of computation that has held sway for 65 or so years, then maybe a fraction of such kudos will rub off on Australia, or even serve to encourage a new generation of Australian computer scientists. More prosaically, this project should attract several PhD students over the years. There is already one working on the conceptual foundations, which has led to publications as well as the stage at which this grant proposal can be made.

### **E6 COMMUNICATION OF RESULTS**

In addition to traditional methods of dissemination of scientific results, viz. Journal publication and Conference presentation etc., this project lends itself to a number of other means of publicity.

- Software artifacts can be made available for general access via the WWW. This would include our collection of Total Functional Programming applications, as well as any new enabling technologies (functional language compilers/interpreters, type checkers, etc.)
- The “revolutionary” aims of the project may even attract some popular media attention.

### **E7 DESCRIPTION OF PERSONNEL**

#### **Chief Investigator**

The sole Chief Investigator will be responsible for the overall conduct of the research, including making key technical decisions on project outcomes, i.e. determination of requirement for Total Functional programs; draft and final formal characteristics of Total Functional programs; selection of modified and new applications for Total Functional Programming; and design of the FP subset language for TFP.

#### **Research Assistant**

To the sought-for Research Assistant will be delegated responsibility for the wide range of programming and technology-related tasks consequent to the above, i.e.: adaptation of existing functional applications for Total Functional Programming; programming of new Total Functional Programming applications; identifying and implementing functional language implementation optimizations; derivation rules for pure & impure platonic combinators; identifying more powerful

type-checker needed for total Functional programs, and if necessary extending a functional language implementation accordingly; programming of simulations for more general analog/systems applications of the concept; adapting existing design recovery tools to generate platonic combinators; assisting with construction of semantic domain and proof techniques.

## E8 REFERENCES

1. Bird, R., "Introduction to Functional Programming", Prentice-Hall (2000)
2. Bailes, P.A., Chorvat, T. and Peake, I., "A Formal Basis for the Perception of Programming as a Language Design Activity", Proc. 1994 International Conference on Computing and Information, Peterborough (1994).
3. Plotkin, G.D., "PCF Considered as a Programming Language", Theoretical Computer Science, 5, pp. 223-255 (1977)
4. Boehm, H. and Cartwright, R. Exact Real Arithmetic: Formulating Real Numbers as Functions, in D.A. Turner (ed.), Research Topics in Functional Programming, pp. 43-64, Addison-Wesley (1990).
5. Hutton, G., "Parsing Using Combinators", Proc. Glasgow Workshop on Functional Programming, Glasgow (1989).
6. Royer, J.S., & J.Case, J., "Subrecursive Programming Systems: Complexity & Succinctness", Birkhauser (1994)
7. Turner, D.A., "Elementary Strong Functional Programming", Proceedings of the first international symposium on Functional Programming Languages in Education, Springer LNCS vol. 1022, pp. 1-13 (1995)
8. Reynolds, J.C. "Three approaches to type structure", in Mathematical Foundations of Software Development, LNCS Vol 185, Springer-Verlag (1985)
9. Hutton, G., "A Tutorial on the Universality and Expressiveness of Fold", Journal of Functional Programming", vol. 9, no. 4, pp. 355-372 (1999).
10. Wadler, P., "Functional Programming in the Real World", URL <http://www.cs.bell-labs.com/who/wadler/realworld/>
11. <http://www.haskell.org/practice.html>
12. Hudak, P., "The Haskell School of Expression – Learning Functional Programming through Multimedia", C.U.P. (2000)
13. Asperti, A., and S. Guerrini, S., "The Optimal Implementation of Functional Programming Languages", CUP (1998)
14. Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 241-250 (1986)
15. Wadler, P., "How to Declare an Imperative", ACM Computing Surveys, vol. 29, no. 3, pp.240-263 (1997)
16. Griswold, R. and Hanson, D.R., "An Alternative to the use of Patterns in String Processing", ACM TOPLAS, vol. 2, no. 2, pp. 153-172 (1980).
17. <http://tfs.cs.tu-berlin.de/~mgr/int02/index.html>
18. <http://www.dcs.shef.ac.uk/~sfedl>
19. Milner, R., "A Theory of Type Polymorphism in Programming", J. Comp. Syst. Scs., vol. 17, pp. 348-375 (1977).
20. Wells, J.B., "Typability and Type Checking in the Second-Order Lambda-Calculus are Equivalent and Undecidable", Logic in Computer Science, pp. 176-185 (1994).
21. Jones, M.P., "First-class Polymorphism with Type Inference", Proc. Symposium on Principles of Programming Languages (1997).
22. Bailes, P.A., Kemp, C.J.M., Peake, I.D and Seefried, S., "Why Functional Programming *Really* Matters", Proceedings 21st IASTED International Multi-Conference on Applied Informatics (AI 2003), pp 919-926, Acta Press (2003).
23. Paulk. M.C., "The Capability Maturity Model for Software: A Tutorial", Lecture Notes in Computer Science, Vol. 750 (1994)
24. Turing, A.M., "On Computable Numbers, with an Application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, No. 42, pp. 230-265 (1936-37).
25. Floyd, R.W., "The paradigms of programming", CACM, vol. 22, no. 8, pp. 455-460 (1979).
26. Backus, J., "Can programming be liberated from the von Neumann style?", CACM, vol. 21, no. 8, pp. 613-641 (1978).