J. Functional Programming 3 (1): 93–115, January 1993 © 1993 Cambridge University Press

An incremental, exploratory and transformational environment for lazy functional programming

COLIN RUNCIMAN, IAN TOYN AND MIKE FIRTH
Department of Computer Science, University of York, Heslington, York YO1 5DD, UK

Abstract

Most programming environments for functional languages offer a single tool used to evaluate programs – either a batch compiler or an interpreter with a read-eval-print loop. This paper presents a programming environment that supports not only evaluation, but also a range of other programming activities including transformation. The environment is designed to encourage working in an incremental and exploratory style, avoiding constraints on the order in which things must be done yet guarenteeing security. What has already been done towards the development of a program automatically persists, as does information about what has yet to be done. For instance, new laws can be introduced as conjectures and used in program transformation, but full details of proof obligations and dependencies are maintained.

The paper outlines the functional language supported by the environment, and uses an extended example to illustrate program construction, execution, tracing, modification and transformation.

1 Introduction

In this paper we describe *two* interactive functional programming systems: *Glide* and *Starship*. They are complementary systems designed to be used in conjunction, and can therefore be viewed as providing a single environment. Each supports the same functional programming language, but the two support different programming activities.

Glide is an interpreter, written in C to run under Unix. It is used primarily to formulate and modify collections of definitions (e.g. of types, functions) and to evaluate expressions. Glide is similar to the interpreters for Turner's languages such as Miranda (Turner, 1985): for example, evaluation is lazy and proceeds by reduction of a fixed set of combinators. The novel aspects of Glide have to do with its lazy incremental processing of programs, its support for an exploratory style of working comparatively free from constraints on the ordering of programming tasks, and the means it provides for sharing and re-using components. For example, a fine-grained incremental type-checking method yields approximate types for those definitions involving as yet undefined free variables; such types are automatically recomputed on the (re-)definition of a relevant free variable in a way that significantly alters type

information. The first version of Glide was developed in 1985, and it has changed little since 1987. It has been used quite extensively, mainly for teaching at York.

Starship is a transformation support system, written in SICStus Prolog (Carlsson and Widen, 1988) to run in the same Unix context as Glide. It is used primarily to reformulate existing definitions so that afterwards some programs will run in less time or space than before, but the results obtained will be identical in all cases. The basic paradigm supported is a variant of the fold/unfold system (Burstall and Darlington, 1977), but with means for defining and proving non-primitive laws and for meta-programming transformational strategies. As in Glide, there are mechanisms to support an incremental and exploratory style of working without compromising security: for example, conjectured laws may be stated and used without proof, but details of proof obligations are maintained and, should a conjecture turn out to be false, there is automatic roll-back of exactly those derivation steps that depended on it. Re-use is supported by the persistence of derivation histories and a replay mechanism. The first version of Starship was built in 1987, and the most recent significant developments of it took place in 1990. It has been used mainly in research projects, but also for teaching in conjunction with Glide.

Section 2 describes the programming language of Glide and Starship as compared with, say, Miranda or Haskell (Hudak and Wadler, 1990). Section 3 presents an extended programming example, illustrating various aspects of our environment. Section 4 outlines some important aspects of the environment that were not illustrated in the previous section. Section 5 discusses related work, and section 6 offers some conclusions and suggestions for future work.

2 Programming language

In the common functional programming language of both Glide and Starship, programs are written in a recursion equation style with non-strict semantics. Programs are subject to a polymorphic type discipline with both primitive and explicitly defined algebraic types. The language does not include any major innovations: we deliberately followed the lead of languages such as KRC (Meira, 1984) and LML (Augustsson and Johnsson, 1987). The result is similar in many respects to a large subset of Miranda or Haskell, so rather than describe the language in detail, we shall only remark on a few design decisions and the reasons for them – especially in view of the kind of environment we wanted to build.

First note some lexical symbols used with different meanings in other functional languages. We use :: for the append operator on lists, because it suggests a compound form of :, the basic list constructor. The @ symbol is used in type declarations to separate expressions from their types, and also in compound symbols such as @ = which are type-related to avoid undue overloading of the symbol =. The infix arithmetic operators /, \ are used to express quotient and remainder, respectively, when one whole number is divided by another; these symbols suggest the complementary relationship between the two operations. The operator for primitive pair construction is ^ (which we pronounce 'hat') in order to make pairs and lists notationally (and aurally) distinct. Finally, the symbol used for equality by definition

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

is → rather than =, because not all defining clauses are true equalities (due to pattern-matching), and because of the essential left-to-right nature of defining clauses from an operational point of view.

The expression language does *not* include special forms such as comprehensions, nor even a conditional since we are content to apply the ***if*** function defined (in a fully explicit style) as follows

```
Define if @ bool→  a→a→a
with   if   True   x   _→x
and    if   False  _   y→y
```

Our reason for omitting special forms is that they tend to obscure the basic applicative nature of expressions and to complicate their manipulation in transformations and proofs. Neither do expressions include lambda abstractions: their anonymity would be a hindrance when it comes to source-level tracing. Expressions do include blocks, and the notation of sections for partial applications of primitive operators – although both these forms of expression proved more troublesome in Starship than we expected.

Only the natural numbers are provided as a primitive numeric type. This avoids the need for special rules (e.g. for integer patterns) and simplifies numeric induction. At one point we even considered making the successor construction non-strict like all other functional constructors: although this has some advantages (Runciman, 1989), efficient evaluation would be harder to achieve, familiar properties such as commutativity of addition would be at risk, and numeric induction would be more tricky. The form of definition for algebraic data types is fairly standard, except that projection functions may be defined along with their complementary constructors: the following example defines ***name @ person→[char]*** and ***age @ person→num*** along with ***Person @ [char]→num→person***:

```
Data person @= Person (name @> [char]) (age @> num)
```

This facility has often proved convenient, particularly as it avoids the need to generate artificial names in some transformations.

The definition language includes constructor-based pattern-matching but not guards. This is because guards would complicate the folding and unfolding of applications quite considerably. Even sequential pattern-matching alone poses all kinds of subtle problems (Firth, 1990).

Finally, there is no notation for modules in the language. Rather, using Glide and Starship as illustrated in the next section, programmers create and maintain *flocks* which serve a similar purpose. This arrangement fits well with the aims of incremental working and an exploratory style. We mention in passing that Glide could be modified to *generate* conventional module notation including suitable import and export lists.

3 Programming example: enumerating the amicable pairs

The *proper factors* of a positive integer ***i*** are the factors less than ***i*** itself. Two distinct positive integers ***i*** and ***j*** are said to form an *amicable pair* if the proper factors of ***i*** sum

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

to *j* and the proper factors of *j* sum to *i*. For example, the pair ***(284,220)*** is amicable because

$$1+2+4+5+10+11+20+22+44+55+110 = 284 \text{ and } 1+2+4+71+142 = 220.$$

In this major section of the paper we shall use the development of a program to enumerate the amicable pairs as an illustrative running example, as we discuss first Glide and then Starship.

3.1 Constructing programs in Glide

Flocks, definitions and types

Typically the development of each application program begins with a Glide session to create a new *flock* within which the top-level definitions for that program are made. A *flock* is a named collection of *definitions* forming a natural unit of work.

$ glide

University of York Glider (built 26 Oct 1989)

Towing ... releasing tow rope ...

glide > **NewFlock amicPairs**

Here we have used the convention of bold font for **user input** and bold italic font for ***computer output***. Note the simple 'teletype' interface with prompts, command lines and responses: our reasons for restricting ourselves to such an interface, and some alternatives to it, will be discussed in section 4.

At the Unix level, flocks are implemented as directories, with definition sources held in files. Some files in a flock are shared – for example, constructor definitions are links to the definition of the parent data type. Programmers do not store or load definition files explicitly, however: these transactions are carried out by Glide so that *persistent definition* and *lazy loading* are automatic. We make the first definition for the amicable pairs program as follows:

glide > **Define main → filter amicable (pairs positives)**

Alternatively, the command ***Edit main*** would have allowed the definition to be formulated using a standard text editor. In either case, the definition persists automatically in the ***amicPairs*** flock until such time as it is explicitly removed – for example, by using an ***UnDefine*** command.

Polymorphic types are maintained incrementally, using a fine-grained algorithm described in detail in an earlier paper (Toyn *et al.*, 1987). So far, none of the names used on the right-hand side of the definition of ***main*** has a definition that is available within the ***amicPairs*** flock: for example, nothing is yet recorded about any possible bindings for ***filter***. The interim type recorded for ***main*** is therefore completely polymorphic. We can use the ***Detail*** and ***Type*** commands to show the state of play

glide > **Detail filter**

glide > **Type main**

a

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

By convention, single letter identifiers are used for universal variables in type formulae, and such variables are named ***a,b,c...*** in order of their first occurrence.

Search sets and Library flocks

Every flock has an associated *search set* containing the names of other flocks whose definitions it may also use. An identifier may be overloaded with several definitions, each belonging to a different flock in the search set. Glide determines the intended binding for each use of an identifer by the type of its context: this type must unify with the type of exactly one candidate definition. Ambiguities are resolved if possible by *forward binding* – binding other identifiers in a way that refines the context type for the overloaded identifier. This search set mechanism with type-based overload resolution is intended to make it easy to share definitions between programs and, perhaps, between programmers. It is important that we have search *sets*, and not ordered search lists or *paths*: binding to the first definition on a path would fail to exploit polymorphic type information; binding to the first definition of suitable type would risk results that depend on the sequence in which bindings are made (Toyn, 1987).

When a flock such as ***amicPairs*** is first created, its search set is empty

glide > **Search**

[ ]

Some flocks are provided as part of the system, forming libraries of components. The most commonly used of these is the flock ⟨***standard***⟩ which may be compared to the *prelude* of other functional programming systems. An ***Edit*** command is used to edit either programmed definitions or search sets. The environment passes control to an ordinary text-editor of the user's choice, post-processing the resulting file. It determines the path-name(s) of the file where the text should be stored for future reference, providing a simple form of automatic persistence.

The ***filter*** function is a ⟨***standard***⟩ higher-order auxiliary. Still within the flock ***amicPairs*** we therefore edit its search set as follows:

glide > **Edit Search**

the initial text

Search [ ]

is altered to

Search [⟨**standard**⟩***]***

Such alteration of the ***amicPairs*** search set does *not* by itself cause any loading of definitions from the secondary storage representation of ⟨***standard***⟩ into primary memory: definitions are loaded only to satisfy a specific need; hence the term *lazy loading*. The programmer is rarely aware of lazy loading taking place because it happens automatically. However, type inference is performed only as definitions are loaded, and does not itself force further loading. In consequence, lazy loading may

refine existing types. Repeating our earlier request for details of ***filter*** is enough to force its definition (but no others) to be loaded from ⟨***standard***⟩

```
glide > Detail filter
Flock: ⟨standard⟩
Type: (a → bool) → [a] → [a]
Source:
-- The items from a list that satisfy a predicate.
-- e.g. filter (2 <) [2,4,1,3] → [4,3]
Define filter @ (a → bool) → [a] → [a]
  with filter p [ ] → [ ]
   and filter p (x:xs) → if (p x) (x:) id (filter p xs)
```

Note the extended application of ***if***. This concise applicative style would be inhibited by a special form for conditionals.

The loading of ***filter*** also causes an incremental refinement of the type for ***main***, and this is further refined by the introduction of a definition for the ***amicable*** function

```
glide > Type main
[a]
glide > Define amicable (i^j) → ami i j & ami j i
glide > Define ami i j → sum (factors i) = j
glide > Type main
[a^b]
```

The auxiliary ***filter*** is just one of the list-processing functions in ⟨***standard***⟩ that seem to find a use in most programs. Others include ***fold***, ***map*** and ***take*** which we shall now use to complete the definition of ***amicable***

```
glide > Define sum → fold (+) 0
glide > Type fold
(a → b → b) → b → [a] → b
glide > Define factors i → filter (factorof i) (take (i − 1) positives)
glide > Define factorof i j → i \ j = 0
glide > Type take
num → [a] → [a]
glide > Define positives → 1 : map (1 +) positives
glide > Type map
(a → b) → [a] → [b]
```

Expression evaluation and snapshots

As in many functional programming systems, the basic nature of a computation in Glide is the evaluation of an expression in the context of some definitions. The

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

environment provides the time-honoured *read-eval-print* loop that has proved so convenient for the simple testing of fresh definitions

glide > **factors 12**

[1, 2, 3, 4, 6]

glide > **amicable (23^45)**

False

The policy of lazy loading continues to apply during evaluation. The obvious benefit is avoidance of unnecessary loading costs; but lazy loading also means we may successfully test the completed parts of incompletely developed programs (in which some definitions have yet to be made, or some type-errors have yet to be resolved) by running the program as a whole with suitably restricted input.

The evaluation strategy is lazy, and the reduction mechanism is based on a fixed set of combinators very similar to those used by Turner (1979) in his seminal implementation of SASL. Computations can be interrupted and examined. This can be done as often as one likes, but only the most recently interrupted computation can be resumed. For example, the definition of ***positives*** can be tested as follows:

glide > **positives**

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,

16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,

29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,

42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,

55, 56, 57, 58, 59, 60, 61, 62, 63 interrupt

At this point, a source-level 'snapshot' of the *remaining computation* may be requested. For historical reasons, this request is made by a ***Trace*** command

glide > **Trace**

map ((+) 1) (63:(positives > > 63))

Snapshots are made possible by introducing at abstraction time (when the source program is translated to combinatory form) special combinators to maintain source-level *annotations* as an integral part of the program graph. This incurs an overhead of about 50% in both space and time. Even with annotations, a partially-reduced combinator graph is somewhat removed from the original source program, so the process of generating a snapshot includes a form of *partial evaluation* eliminating compiled combinators as far as possible. We gave details of the method in an earlier paper (Toyn and Runciman, 1986).

After a computation has been interrupted, normal reduction can be resumed. Moreover, if a snapshot has been produced, it is the partially evaluated graph that is used for the continuing computation: the snapshot-time reductions do not have to be repeated under a different strategy

glide > **Ok**

, 64, 65, 66,...

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

Binding and type errors

Returning to the overall aim of developing a program to enumerate the amicable pairs, we have yet to define the ***pairs*** function, as we may discover by trying to compute ***main***

glide > **main**

Undefined variable: pairs

Used by: amicPairs|main

Context type: [num] → [num^num]

To provide a context for a discussion of type errors, we shall now make a deliberately erroneous definition

glide > **Edit pairs**

text editor used to create

Define pairs [ ] → [ ]

and pairs (x : xs) → map x ^ xs pairs xs

Although this is clearly type-incorrect (e.g. ***xs*** cannot be both a list and a function), the only response is a change of prompt to include ***E*** (for error) as a warning

glide E >

Think of this warning signal as an approximation, on a dumb ASCII terminal, to a small amber light in one corner of the screen.

Sources in which type-errors are detected are annotated with the details, and placed by the system in an internal list of definitions with unresolved errors. Experience has taught us that some such muted or delayed response to errors is an important ingredient in an incremental system, because an error detected in one definition is often corrected by a change to another definition *which the user already had in mind.* If errors are resolved, the warning signal is turned off and the relevant definitions are taken out of the error list with error annotations removed.

Requesting ***Edit*** with no argument yields an editor applied to some definition in the error list. Each problematic expression has a simple label (a capital letter): this is used in comment lines, both to delimit the expression and to introduce a footnote giving details of the problem

glide E > **Edit**

the annotated text looks like this

Define pairs [ ] → [ ]

and pairs (x : xs) → map x ^ xs pairs xs

--? A BB A

--? A..A @ [a] → b^c but context @ [d] → [e]

--? B..B @ [a] but context @ f → g → c

This particular format for type error messages was by no means the first we

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

implemented. At one time, for example, the system used sophisticated pretty-printing of definitions to allow embedded messages: we hoped that this would be particularly helpful in revealing the true structure of expressions where mistaken operator associations or precedences lay at the root of type errors, but users (beginners especially) only protested that they could no longer recognise their own definitions! Pretty-printing of definitions is now applied *only* on explicit request.

The above definition of ***pairs*** becomes well-formed with the addition of a couple of brackets and an append operator

Define pairs [ ] → [ ]

and pairs (x : xs) → ***map*** (x^) xs :: ***pairs*** xs

However, this is not quite what we want for the amicable pairs program. It works well for finite lists

glide > **pairs "socks"**

[('s'^'o'), ('s'^'c'), ('s'^'k'), ('s'^'s'), ('o'^'c'),
('o'^'k'), ('o'^'s'), ('c'^'k'), ('c'^'s'), ('k'^'s')]

but it is not much use for infinite ones

glide > **pairs positives**

[(1^2), (1^3), (1^4), (1^5), (1^6), (1^7), (1^8),...

If we think of the pairs (x,y) as occupying an infinite $x-y$ co-ordinate grid, then in this order of enumeration we never get beyond the first column in which $x = 1$ in every pair. What is required is a *diagonalised* enumeration

glide > **Edit pairs**

we amend the definition to the following

Define pairs →

(***pairs'*** [ ] ***where***

pairs' xs (y : ys) → ***map*** (y^) xs :: ***pairs'*** (y : xs) ys)

glide > **pairs positives**

[(2^1), (3^2), (3^1), (4^3), (4^2), (4^1), (5^4),...

This completes our prototype program to enumerate the amicable pairs.

Type declaration

We have formulated all the definitions for the program without any explicit type declarations, but could have included these had we wished. In other functional programming systems that allow polymorphic type declarations, declared types must be instances of inferred types. In Glide, because of the incremental nature of type-checking, we require rather that declared types should *unify* with inferred types: the *actual* type of a definition is taken to be the result of that unification. The system can be requested to annotate definitions with type declarations that reflect all that can be

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

inferred about their type at that stage of development. If we now do this for the amicable pairs program

glide **> TypeAnnotate main**

the result obtained is shown in Fig. 1.

```
Define ami @ num → num → bool
  with ami i j → sum (factors i) = j
Define amicable @ num^num → bool
  with amicable (i^j) → ami i j & ami j i
Define factorof @ num → num → bool
  with factorof i j → i \ j = 0
Define factors @ num → [num]
  with factors i →
      filter (factorof i) (take (i−1) positives)
Define main @ [num^num]
  with main → filter amicable (pairs positives)
Define pairs @ [a] → [a^a]
  with pairs →
      (pairs' [ ]
      where pairs' xs (y:yx) →
          map (y^) xs :: pairs' (y : xs) ys)
Define positives @ [num]
  with positives → 1 : map(1 +) positives
Define sum @ [num] → num
  with sum → fold (+) 0
```

Fig. 1. Initial ***amicPairs*** definitions after type annotation.

Running the program

To run the program, we simply evaluate the expression ***main***. The example amicable pair given earlier is the first encountered in the enumeration order of ***pairs positives***

glide **> main**

[(284,220)...

It is rather a long wait for each pair! This has a lot to do with the specification-like programming style we have adopted, but it is also true that the Glide interpreter is quite slow. Speed of execution was never a major aim. However, the system does provide a handle for alternative methods of implementing evaluation: FLIC intermediate code (Peyton Jones, 1988) can be generated on request, and we have separate tools to compile this via G-machine (Johnsson, 1984) or TIM (Fairbairn and Wray, 1987) code, for example.

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

With our initial version of the amicable pairs program now complete, we quit Glide in the terse Unix style, by keying end-of-file

glide **> ^D**

The ***amicPairs*** flock, its search set, and the definitions it contains, all persist for use in another Glide session or, as we are about to see, in Starship.

3.2 Transforming programs in Starship

As we said in the introduction, Glide and Starship are separate programs, written in different languages. They communicate only by the creation and modification of textual source files in which programmed definitions persist. Each system also maintains files of private information: Starship needs these to check, for example, that no other program (such as Glide) has been used to alter a definition which is the subject of a proof or transformation only partially completed in a previous session.

To apply Starship to the amicable pairs program, we begin by locating the appropriate flock

$ starship

University of York Starship

Version 0.2 launched 22 January 1990

Beaming up... welcome aboard...

> Flock amicPairs

Simple in-lining

Let us begin with a minor transformation to unfold an application in-line. Recall that the function ***factorof*** is used only in the definition of ***factors***: the relevant definitions can be listed as follows:

> List factors factorof

factorof @ num → num → bool

1. factorof i j → i \ j = 0

factors @ num → [num]

1. factors i → filter (factorof i) (take (i − 1) positives)

The suppression of keywords such as ***with***, ***Define*** and ***and*** in favour of *clause numbers* reflects the different nature of programming in Starship as compared with Glide. No ***Edit*** command is provided: it would hardly be safe! Yet we must manipulate expressions and clauses: expressions are specified mainly by patterns they match, and clauses by the name of what is defined, qualified by a number if necessary. There are sensible defaults.

Partial applications, such as that of ***factorof*** in ***factors***, cannot be unfolded directly in the current Starship system. The curried nature of ***factorof*** must first be made explicit by reducing the arity of its definition. The operator-operand form ***i \ j = 0*** is

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

just a shorthand for the applicative form *(=) ((\) i j) 0*, which can be re-expressed using (*o*) the functional composition operator

```
> Fold (o) in factorof
factorof @ num → num → bool
*1. factorof i j → ((= 0) o (i\)) j
> Retract
factorof @ num → num → bool
*1. factorof i → (= 0) o (i\)
```

Of course, we should like to automate such steps, but we also want source-level definitions accessible to the user at all times. Simple schemes based on combinatory coding do not give acceptable results in general. Note the *-markers introducing clauses after transformation steps: these indicate clauses that have changed as a result of the last step.

With a reduced arity definition, the unfold step is easy

```
> Unfold factorof in factors
factors @ num → [num]
*1. factors i → filter ((= 0) o (i\)) (take (i−1) positives)
```

We can quit Starship at this point with no outstanding proof obligations; the system reminds us which definitions have been transformed

```
> ^D
Transformed in amicPairs: factorof factors
End of SICStus execution, user time 1.820
```

The new definitions are now in force. Also, the old ones from which they were derived are retained under the RCS version control system (Tichy, 1985) unless the user has opted to switch off this mechanism. Use of RCS not only provides a useful record of developments for the programmer, it also supports an ***Undo*** command that really can undo *any* previous step.

Specialisation by case analysis

There are many different transformation strategies possible. In their original fold/unfold paper, Burstall and Darlington emphasised a strategy of *specialisation* by considering cases: their method begins with the introduction of new instances of existing clauses with more specific argument patterns. We have discussed in a previous paper (Runciman *et al.*, 1990) the necessary adaptation of instantiation, and related rules, for a language with non-strict constructors and lazy matching.

Using a theory of tabulation

In more recent years, there has been a shift towards the definition and use of *equational theories* in transformation. These theories are collections of equivalence laws, some perhaps with side conditions, relating a particular group of functions.

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

Bird's (1988) theory of list operators is an outstanding example: we could sensibly apply *filter promotion*, for instance, to our amicable pairs program.

We choose instead to illustrate the definition and use in Starship of a single-law theory of *lazy tabulation* – an idea originally due to Turner (1981). Given a function *f* over the natural numbers we can tabulate (or memoise) its results in an infinite list, replacing all other applications of *f* by a linear indexing operation on this table. Since the spine and items of the table are evaluated lazily, only those parts of it that are actually needed will be computed. To formalise this idea in Starship, we first define what we mean by the naturals, a table of results for a function of the naturals, and the linear indexing operation

> Define naturals → 0: map (1 +) naturals

> Define table f → map f naturals

> Define index (x:_) 0 → x

and index (_:xs) (n + 1) → index xs n

It remains to formulate the law, and in due time to prove it. A law declaration is similar in form to a single-clause function definition: instead of ***Define*** we write ***Law***, instead of a function name there is a law name, and instead of argument patterns there may be variables of the law – implicitly, these are universally quantified. On the right hand side of the → symbol is an equivalence between expressions, or the entailment of an equivalence subject to one or more other equivalences as side-conditions. The exclusive emphasis on equivalences reflects the intended transformational application of laws. Expressions in laws may involve the undefined value ⊥.

> Law tabulation f → f ⊥ ≡ ⊥ ⊢ f ≡ index (table f)

For tabulation to apply, *f* must be strict, because the indexing operation is strict. This is just the sort of side-condition that is easily overlooked, which is one motivation for building a transformation support system with a proof-checker.

In keeping with the exploratory style, ***tabulation*** can now be applied, even though it has not yet been proved. Consequent proof obligations and dependencies are recorded automatically. In the amicable pairs program, sums of factors are computed *infinitely often* for each number, suggesting that tabulation might be worthwhile! Hence we proceed as follows:

> Fold (sum o factors) in ami

***1. ami i j → (sum o factors) i = j**

> Apply tabulation (sum o factors)

***1. ami i j → index (table (sum o factors)) i = j**

The side-condition requiring ***(sum o factors)*** to be strict is verified automatically. In other law applications, it may not be possible to deal with a side-condition automatically, in which case it is retained for subsequent explicit proof. Starship incorporates several strictness and finiteness analysis techniques to avoid the proliferation of such proof obligations.

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

As yet, we may appear only to have made things worse, adding the machinery of table lookup to ***ami*** without removing the composite function. To make the intended gain, it is essential that a single table of results for this function is shared across all applications of ***ami***. A clever compiler might achieve this for us by lifting the ***table*** application out of ***ami***, recognising it as a *constant applicative form* (Peyton Jones, 1987); but Glide does not perform such optimisations. We therefore make the sharing explicit in a final transformation step at source level

> Define factorsums → table (sum o factors)

> Fold factorsums in ami

1. ami i j → index factorsums i = j

Although the necessary transformation of defining clauses is now complete, the equivalence of the new definition of ***ami*** to the original one relies on the ***tabulation*** law, which we have yet to prove. We can request information about all such outstanding dependencies using a ***Status*** command

> Status

amicPairs|ami: tabulation

Proving laws

When a law is defined, a *proof clause* is automatically created for it. Initially the proof clause is just a copy of the law itself

> Proof tabulation

Proof clauses of law tabulation

1. f⊥ ≡ ⊥⊢f ≡ index (table f)

The Starship proof checker supports a backward style of proof in which proof clauses are transformed to ***TRUE***. Most commands used for transforming clauses of definitions are also applicable to proof clauses. We begin the proof of ***tabulation*** by exposing a little of its structure

> Unfold table

Proof clauses of law tabulation

1. f⊥ ≡ ⊥⊢f ≡ index (map f naturals)

> Extend with n

Proof clauses of law tabulation

1. f⊥ ≡ ⊥⊢f n ≡ index (map f naturals) n

Proof of lemmas

Just as Glide supports top-down program development, so Starship supports top-down proof by conjecturing other laws as lemmas. One problem-solving strategy we have often found successful in this context is *difference reduction* (Ernst and Newell, 1969) – repeatedly introducing and applying lemmas whose application removes a difference between left and right hand sides in the main equivalence being proved.

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

Using this strategy, the first lemma in the ***tabulation*** proof raises ***f*** to the outermost position on the right hand side, as on the left

```
> Law indmap f xs n→f⊥ ≡ ⊥⊢f (index xs n) ≡ index (map f xs) n
> Apply indmap f
Proof clauses of law tabulation
*1. f⊥ ≡ ⊢f n ≡ f (index naturals n)
```

As recorded above, the ***Apply*** command is ambiguous: in which direction should the law be applied? What actually happens is a screen-based interaction not easy to show here: the system highlights each possible expression to which the law could apply in turn; the user skips over candidates using the space bar and accepts one using the return key. This simple mechanism is perhaps the most elaborate in our deliberately modest, terminal-based interface.

The second and final lemma in the ***tabulation*** proof just equates arguments in the left and right hand side applications of ***f***

```
> Law indnat n→index naturals n ≡ n
> Apply indnat
Proof clause 1.tabulation is TRUE
Law tabulation is provisionally proved
```

The proof is only provisional because the lemmas ***indnat*** and ***indmap*** have yet to be proved

```
> Status
Law amicPairs/tabulation: indnat indmap
```

Proof by induction

The Starship system incorporates a straightforward structural induction scheme which is generic over almost all data types. The one special case is natural induction, where finiteness information is used to improve the scheme. Although fixpoint induction is not directly supported, some useful lemmas whose proofs would require fixpoint induction are available in a standard library – for example, the *take lemma* (Bird and Wadler, 1988).

We can prove ***indnat*** by natural induction, and ***indmap*** by list induction

```
> Proof indnat
Proof clauses of law indnat
  1. index naturals n ≡ n
> Cases n
Proof clauses of law indnat
*1.1. index naturals ⊥ ≡ ⊥
*1.2. index naturals 0 ≡ 0
*1.3. index naturals n1 ≡ n1 ⊢
      index naturals (n1+1) ≡ n1+1
```

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

Since Starship infers the type of ***n*** to be natural, the choice of induction scheme in response to ***Cases n*** is automatic. The clause numbering scheme generalises in the obvious way.

The base cases in this inductive proof are easily dealt with

> Unfold in 1.1 1.2

Proof clause 1.1 . indnat is TRUE

Proof clause 1.2 . indnat is TRUE

Such an ***Unfold*** command with no specific parameters requests exhaustive unfolding, subject only to the need for termination which is ensured by a variant of pending analysis (Young and Hudak, 1986). A few basic simplifications, such as equivalence of identical expressions, are performed automatically after each proof step.

If we try taking the same ***Unfold*** approach to clause ***1.3***:

> Unfold in 1.3

Proof clauses of law indnat

$$*1.3.\ index\ (0: map\ (+1)\ naturals)\ n1 \equiv n1 \vdash$$
$$index\ (0+1: map\ (+1)\ (map\ (+1)\ naturals))\ n1 \equiv n1+1$$

the effect is excessive. In such circumstances ***Undo*** is invaluable. Without a full implementation of ***Undo***, exploratory use of composite transformation steps would inevitably be inhibited by the risk of regret

> Undo

Unfold in 1.3 . indnat undone

Having backed out of the blanket ***Unfold*** approach, we now advance again more cautiously, requesting a limited unfolding of a selected ***index*** application only. Argument patterns in the command are used to specify which application we intend to unfold

> Unfold index _ (n1 + 1) in 1.3

Proof clauses of law indnat

$$*1.3.\ index\ naturals\ n1 \equiv n1 \vdash$$
$$index\ (map\ (1+)\ naturals)\ n1 \equiv n1+1$$

Now we spot an opportunity to apply the other lemma. Fully specific argument patterns specify the target expression, and once again a strictness side-condition is dealt with by the system

> Apply indmap (1 +) naturals n1

Proof clauses of law indnat

$$*1.3.\ index\ naturals\ n1 \equiv n1 \vdash$$
$$1+index\ naturals\ n1 \equiv n1+1$$

Since Starship uses *AC-matching* with respect to primitive operators (Hullot, 1979), and + is commutative, it only remains to take the inductive step. In general, this

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

requires the selection of both an assumption and a target expression; the default method uses enumeration of candidates by highlighting expressions in place, as described earlier

> **Induce**

Proof clause 1.3 . indnat is TRUE

Law indnat is provisionally proved

> **Status**

Law amicPairs/indnat: indmap

Law amicPairs/tabulation: indnat indmap

The ***indmap*** proof is a little more complex, but the mechanisms involved are very similar, so we omit the details.

4 Aspects of the environment not illustrated by the example

Choosing to describe the environment by means of an example, as in the previous section, has the inevitable consequence that not all aspects are illustrated. The fullest accounts of the Glide and Starship systems, including various implementation details, can be found in the second and third authors' DPhil theses. However, we should like to outline here one or two important aspects of each system that we have not yet discussed.

4.1 More about Glide

Alternative user interface

Despite the simplicity and ease of learning of the teletype command interface of Glide, it is lacking in some respects. A user trying to resolve a type inconsistency, for example, may want to inspect the types of sub-expressions near those marked as inconsistent: this information is held by the underlying system, but cannot be accessed conveniently by the user. Similarly, ***Edit*** gives access to only one definition at a time, since context is lost from the screen when the editor is invoked.

Therefore, still working only with a terminal rather than a workstation (because many of our student users did not have workstations), we developed a 'full-screen' version of Glide. The user can select any sub-expression in a definition by traversing syntactic structures using the arrow keys and another key meaning 'expand selection to become the enclosing expression'. The type of any selected expression may be inspected. If an identifier is selected its definition can be brought onto the screen, so several related definitions may be viewed at once. The definitions being viewed are stacked: a fresh request to view a definition pushes it onto the stack (if it is already in the stack, it moves to the top). The screen displays as many definitions from the top of the stack downwards as will fit: the small size of a terminal screen is restrictive, but most definitions are smaller still. Space is left above the stack for responses to requests for information such as types.

Definitions can be thought of as the nodes, and bindings between them as the links, of a *hypertext* system. In such systems, knowing where you are, how you got there, and how to get to somewhere else, can be tricky problems. In full-screen Glide,

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

various *menus* of definitions are available such as: all definitions with views currently on the stack; those in which errors have been detected; those that the top definition uses; those that use the top one (among the definitions loaded so far). Similar menus are available for flocks. Programming commands too are available in a menu structure but there are also single key-strokes for most of them. A ***what_is*** key requests information about any menu entry.

Structure editing of a terse, expression-based language is painful, and we therefore decided to provide a simple text editor within the system. This led to each definition on the stack being in one of two modes: browsing or editing. To give some idea of the edit mode, commands available in addition to modeless character insertion include *mark*, *copy*, *cut*, *paste*, *erase-character*, *erase-line* and *match-bracket*. Edits to one definition need not be completed before others can be browsed or edited, but an edited definition must parse cleanly before it can itself be browsed structurally.

Expression evaluation is the most awkward part of Glide to integrate into the full-screen system. An expression to be evaluated is formulated in a new top stack frame as a named definition. One benefit of this is that it persists for repeated use as a test case. Results (and any tracing information) simply scroll up the screen from the bottom.

Our experience has been that full-screen Glide is a big improvement for understanding and re-structuring larger programs: it has much greater power for these tasks. The twin modes do complicate the dialogue structure, but they seem no worse than the context-switching of ***Edit*** in the teletype interface. For evaluating expressions, however, it is hard to beat the real-eval-print loop: having to create a named definition on the display stack for each main expression feels too much like hard work.

4.2 More about Starship

Higher level transformations

The rules of the fold/unfold system operate at a rather low level. By supporting the definition and proof of laws relating non-primitive functions we potentially raise this level significantly. In Starship, any group of laws that forms a *convergent rewriting system* (Huet and Oppen, 1980) can also be declared as a named ***Lawset*** and selectively applied as a simplification procedure. Such a simplification procedure for the primitives is built in as part of the system. But the basic rules, even augmented by laws and simplification procedures, must still be applied according to some higher level strategy.

The seminal work on LCF (Gordon *et al.*, 1979) solved this problem in the context of theorem proving by *programming* strategies in a *meta-language*, using higher order functions to compose steps and a type-checker to guarantee the soundness of computed proofs. Like others, we have followed this lead.

ASTRAL (A STRAtegy Language) is our transformational meta-language with primitives corresponding to the interactive commands of the Starship system and various combining tactics as primitive operators. It is a lazy functional language and hence supports backtracking (Wadler, 1985). Among the more distinctive features of ASTRAL are: a comprehension mechanism which abstracts away from associative

and commutative rearrangements of expressions; a memo-response method and a continuation-based method for expressing interactive transformations involving both programmer and machine, without loss of referential transparency; and a system of abstract data types that supports the convenient expression of transformations affecting only a small part of a program leaving the rest unchanged. A prototype implementation is complete apart from continuation-based interaction. We hope to give a detailed description of ASTRAL and its implementation in a future paper.

Selective undoing and replay

As noted during the example, our experience has confirmed the need for a secure and convenient way of undoing earlier transformation steps. We stress that the ***Undo*** command in Starship, and accelerated forms of it using ***Mark*** and ***Back*** commands, have proved invaluable. However, they amount only to requests for a global retreat to some earlier point in session history: sometimes this is too indiscriminate, as many 'good' steps may have been interleaved among the 'bad' ones. So in the most recent version of Starship we introduced two further mechanisms.

First, there is a selective undo mechanism which uses precisely the same dependency and roll-back scheme as that for laws, law applications and proofs. An arbitrary name can be declared as an ***Idea*** and applied to definition or proof clauses. An idea is like an unformulated law in the programmer's head. Applying an idea to a clause has no effect that is immediately apparent, but all subsequent versions of that clause depend on the validity of the idea. Ideas are 'proved' simply by naming them in a ***Confirm*** command, or 'disproved' by naming them in an ***Abandon*** command.

The other provision is that a user has the option not only to record previous versions of a program but also to record the steps used to transform one version to another. These recorded derivations can be used in a subsequent ***Replay*** command, applying the same transformation to the same initial definitions but with the opportunity to omit, insert, replace or systematically modify steps as the transformation proceeds. Moreover, recorded transformations need not necessarily be re-applied to the same initial definitions, but perhaps to another program with similar structure. Implementing ***Replay*** presented few difficulties – another advantage of using a simple text-based interface.

5 Related work

Since the seminal developments in the mid-1970s, represented by Burstall's (1977) NPL and Turner's (1976) SASL, there have been many different implementations of functional programming in the recursion equation style. Higher order functions are almost universally supported, and in this connection the discipline of polymorphic typing has gained widespread acceptance. Although other forms of incremental polymorphic typing have been implemented (Nikhil, 1985), the combination of lazy loading with fine-grained re-checking is unique to Glide so far as we know. Most *purely functional* systems now employ lazy evaluation reflecting a language with non-strict semantics (Haskell and Miranda are prominent examples), but in systems implementing a *mostly functional* language such as standard ML (Milner *et al.*, 1990)

eager evaluation is preferred because sequential imperatives can be understood in the context of applicative evaluation order. Whereas sophisticated debugging tools have been implemented for ML by automatic instrumentation of source code (Tolmach and Appel, 1990), lazy evaluation makes it difficult to provide a satisfactory equivalent, but O'Donnell and Hall (1988) describe one approach. Although Glide's snapshot mechanism is rather limited, at least one variant of it has been developed (Snyder, 1990).

The few functional language implementations apart from Glide and Starship that have included support for program transformation have generally been based on strict languages with applicative order evaluators. For example, Feather (1982) describes his ZAP system for the fold/unfold transformation of NPL programs, in which the emphasis is on automated tactics guided by a heuristic meta-program. The same emphasis is apparent in a series of systems developed by Darlington *et al.* (1989), culminating in a transformational programming environment for the Hope+ language that provides several sophisticated algebraic transformations, such as linearisation and inversion, as primitives of a meta-language. There are many differences between the Hope+ transformation system and Starship. As to the programming language, for example, Hope+ includes *absolute set abstraction* (ASA). This is a powerful special form with no immediate applicative equivalent: programs involving ASAs can be directly executed only by a specialised search procedure based on *narrowing*; a major motivation for the whole environment is to support the transformation of ASAs into conventional functional programs. However, the particular difference between the two systems we should like to emphasise again concerns the exploratory style. In Starship the primary mode of working is interactive, with conjectured laws, roll-back, and so on; even ASTRAL meta-programming is strongly biased towards exploratory interaction, and meta-programs yield lazy lists of alternative results. In Hope+ transformations the primary emphasis is on the application of prepared procedures, and meta-programs yield a single resulting product of transformation.

6 Conclusions and future work

Sometimes a distinction is made between programming environments that are rigorous (constraining how the programmer works so that everything can be checked) and those that are exploratory (leaving the programmer to work free-style but at their own risk). In the Glide and Starship systems we have tried to achieve rigour in an exploratory style, and we claim to have succeeded in many respects.

Structuring the environment as a pair of co-operating tools, internally very different but sharing a common source language for communication, has worked surprisingly well. Using source-level text files to represent an automatically persisting state of the environment is straightforward, but hardly efficient: even the most heavily used definitions in the ⟨***standard***⟩ flock are processed from source by each user in each session (subject to lazy loading, of course). Work on a new version of the environment, with a single *definition server* shared among all users, is at a preliminary stage.

https://doi.org/10.1017/S0956796800000629 Published online by Cambridge University Press

Many aspects of the present environment are linked to the type system, from overload resolution in Glide to induction schemes in Starship. There are still further ways of exploiting type information that we have yet to implement fully – for example, context-based retrieval from libraries (Runciman and Toyn, 1991). Type information could also be used to automate the validation of a large class of laws (Wadler, 1989) and further refinements of the complex algorithms used for incremental type inference are still being discovered (Aditya and Nikhil, 1991).

The problem of how best to *observe* a lazy functional computation in progress is still open. Glide's snapshots of partially reduced program graphs provide the information required for some purposes, but they are too static (and often too large) to give the programmer an adequate view of what is going on. There is ample scope for experiment, for example, with presentations of a series of program graphs in some highly simplified or condensed form, filtering out a mass of information that is irrelevant for particular purposes: at least one system has been implemented with a mechanism for producing such representations (Taylor, 1991).

Is it not surprising that so few implementations of functional programming languages include transformation support, since ease of formal manipulation is so often claimed? Building the Starship system has been instructive in this respect: though some things turned out to be easier than we expected, most turned out to be far more difficult. Starship is a complex program of several thousand lines, and for many parts of it we have no explicit formalisation other than the program itself. We share Feather's (1987) goal of achieving a symbiosis between the talents of the user (who must take some strategic decisions) and the machine (which must carry out numerous low-level manipulations flawlessly). But in Starship, even extended with ASTRAL, the communication level between human and system remains low. It is hard to compensate for the limits in the programmer's ability to understand and work with a program structure which, as transformation proceeds, changes, and becomes more complex; the more powerful the mechanical system, the worse this problem is. We stress the need for near-perfect formatting of programs, clear marking of freshly transformed expressions, access to previous versions and a comprehensively flexible way to revise earlier steps; but this can only be the beginning.

Finally, there is no formal guarantee in Starship that a transformed program is more efficient than its predecessor. There are many reasons why this is so, but the dominant one is that we do not yet have a suitable model of the space and time costs of lazy evaluation. A practical consequence is that although transformational development may eliminate a good deal of empirical testing otherwise necessary to check program *correctness*, evaluation under a specific implementation (such as Glide) is still needed to assess the *relative efficiency* of semantically equivalent programs.

Acknowledgements

Various suggestions made to us by others have directly contributed to the development of Glide and Starship: these have come from Alan Dix, Janet Finlay, Sandra Foubister, John Howard, Nigel Jagger, Mike Johnson, Jagdip Patel, Paul Sanders, David Stokes, David Wakeling, and perhaps others to whom our apologies

are due for their anonymity here. Hundreds of students have used the Glide and Starship environment and patiently mailed us about their difficulties and wishes. Our thanks go to Phil Wadler for his comments on earlier material on which this paper is based, and to the referees for their suggestions. Financial support for our work has come from the Science and Engineering Research Council of Great Britain and from British Telecommunications plc.

References

Aditya, S. and Nikhil, R. S. 1991. Incremental Polymorphism. In *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, Springer-Verlag, pp. 379–405.

Augustsson, L. and Johnsson, T. 1987. LML Users' Manual. PMG Report, Department of Computer Science, Chalmers University of Technology.

Bird, R. 1988. Lectures on constructive functional programming. Technical Monograph PRG-69, Programming Research Group, Oxford University.

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice-Hall.

Burstall, R. M. 1977. Design considerations for a functional programming language. In *The software revolution: state-of-the art conference*, Pergamon Press, pp. 45–57.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *J. ACM*, **24** (1): 44–67.

Carlsson, M. and Widen, J. 1988. SICStus Prolog users, manual. Technical Report R88007, Swedish Institute of Computer Science.

Darlington, J. et al. 1989. A functional programming environment supporting execution, partial execution and transformation. In *PARLE '89, Parallel Architectures and Languages Europe*, Springer-Verlag, pp. 286–305.

Ernst, G. and Newell, A. 1969. *GPS: A Case Study in Generality and Problem Solving*. Academic Press.

Fairbairn, J. and Wray, S. 1987. TIM: a simple, lazy abstract machine to execute supercombinators. In G. Kahn (editor) *Functional Programming Languages and Computer Architecture*, Springer-Verlag, pp. 34–45.

Feather, M. S. 1987. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, **4** (1): 1–20.

Feather, M. S. 1987. A survey and classification of some program transformation approaches and techniques. In L. G. L. T. Meertens (editor) *Program Specification and Transformation: Proc. IFIP TC2/WG2.1 Working Conf.*, North-Holland.

Firth, M. A. 1990. A Fold/Unfold System for a Non-strict Language. DPhil Thesis YCST 90/09, Department of Computer Science, University of York.

Gordon, M. J. C., Milner, R. and Wadsworth, C. P. 1979. *Edinburgh LCF*. Springer-Verlag.

Hudak, P. and Wadler, P. (Eds.), 1990. Report on the Programming Language Haskell. Technical Report, Yale University and Glasgow University.

Huet, G. and Oppen, D. C. 1980. Equations and rewrite rules – a survey. In *Formal Language Theory – Perspectives and Open Problems*. Academic Press, pp. 349–405.

Hullot, J. M. 1979. Associative commutative pattern matching. In *Proc. 6th International Joint Conference on Artificial Intelligence*, pp. 406–412.

Johnsson, T. 1984. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, **19** (6): 58–69.

Meira, S. L. 1984. *The Kent Recursive Calculator (KRC): Syntax and Semantics*. Computing Laboratory, University of Kent.

Milner, R., Tofte, M. and Harper, R. 1990. *The Definition of Standard ML*. MIT Press.

Nikhil, R. S. 1985. Practical polymorphism. In *Proc. 2nd ACM Conf. on Functional Programming Languages and Computer Architecture*, Springer-Verlag, pp. 319–333.

O'Donnell, J. T. and Hall, C. V. 1988. Debugging in applicative languages. *Lisp and Symbolic Computation*, **1** (2).

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall.

Peyton Jones, S. L. 1988. FLIC – a Functional Language Intermediate Code. *ACM SIGPLAN Notices*, **23** (8): 30–48.

Runciman, C. 1989. What about the *natural* numbers? *Computer Languages* **14** (3): 181–191.

Runciman, C., Firth, M. and Jagger, N. 1990. Transformation in a non-strict language: an approach to instantiation. In *Proc. Glasgow Workshop on Functional Programming*, Springer-Verlag, pp. 133–141.

Runciman, C. and Toyn, I. 1991. Retrieving reusable software components by polymorphic type. *J. Functional Programming* **1** (2): 191–211.

Snyder, R. M. 1990. Lazy debugging of lazy functional programs. *New Generation Computing*, **8** (2): 139–161.

Taylor, J. 1991. A system for representing the evaluation of lazy functions. Technical Report 522, Department of Computer Science, Queen Mary and Westfield College, University of London.

Tichy, W. F. 1985. RCS – a system for version control. *Software – Practice and Experience*, **15** (7): 637–654.

Tolmach, A. P. and Appel, A. W. 1990. Debugging standard ML without reverse engineering. In *Proc. 6th ACM Conf. on LISP and Functional Programming*, pp. 1–12.

Toyn, I. and Runciman, C. 1986. Adapting combinator and SECD machines to display snapshots of functional computations. *J. New Generation Computing*, **4**: 339–363.

Toyn, I. 1987. Exploratory Environments for Functional Programming. DPhil Thesis YCST 87/02. Department of Computer Science, University of York.

Toyn, I., Dix, A. and Runciman, C. 1987. Performance Polymorphism. In *Proc. 3rd ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 325–346.

Turner, D. A. 1976. SASL language manual. Technical report, Department of Computational Science, University of St Andrews.

Turner, D. A. 1979. A new implementation technique for applicative languages. *Software – Practice and Experience* **9** (1): 31–49.

Turner, D. A. 1981. The semantic elegance of applicative languages. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM Press, pp. 85–92.

Turner, D. A. 1985. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture*, Volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 1–16.

Wadler, P. 1985. How to replace failure by a list of successes. In *Proc. 2nd ACM Conf. on Functional Programming Languages and Computer Architecture*, Springer-Verlag, pp. 113–128.

Wadler, P. 1989. Theorems for free! In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM Press, pp. 347–359.

Young, J. and Hudak, P. 1986. Finding Fixpoints on Functional Spaces. RR-505, Department of Computer Science, Yale University.