# Contents

1 Proof Support for General Type Classes ............................................. 1  
   Ron van Kesteren, Marko van Eekelen, Maarten de Mol
   1.1 Introduction ........................................................................ 1  
   1.2 Sparkle ............................................................................. 4  
   1.3 Preliminaries .................................................................... 5  
   1.4 Structural induction .......................................................... 7  
   1.5 Induction on instances ....................................................... 8  
   1.6 Multiple class constraints .................................................. 10  
   1.7 Implementation ................................................................ 14  
   1.8 Related and future work ..................................................... 14  
   1.9 Conclusion ....................................................................... 15  
   References ........................................................................... 15

2 Generic Proofs for Combinator-based Generic Programs ....................... 17  
   Fermín Reig
   2.1 Introduction .................................................................... 17  
   2.2 ‘Boilerplate’ combinators as generic functions ..................... 19  
   2.3 Generic proofs for ‘boilerplate’ combinators ....................... 20  
   2.4 A proof of the fusion law for *gmapF* ................................ 21  
   2.5 A theorem about *occurs* .................................................. 22  
   2.6 A fusion law for *everywhere* ............................................. 25  
   2.7 Conclusions and further work .............................................. 28  
   2.8 Proof of lemma 2.1 ............................................................ 29  
   2.9 Proof of lemma 2.9 ............................................................ 29  
   2.10 Proof of lemma 2.3 ........................................................... 30  
   2.11 Proof of theorem 2.5 (continued) ...................................... 30  
   References ........................................................................... 31

3 Building certified components within FOCAL ....................................... 33  
   Catherine Dubois, Thérèse Hardin, Véronique Viguié Donzeau Gouge
   3.1 Introduction .................................................................... 33  
   3.2 FOCAL requirements and design principles ....................... 34  
   3.3 Overview of FOCAL ......................................................... 35
## Calculating an Exceptional Machine

**Graham Hutton, Joel Wright**

- **4.1 Introduction** ........................................... 49
- **4.2 Abstract machines** ....................................... 50
- **4.3 Arithmetic expressions** ................................. 51
- **4.4 Adding exceptions** ....................................... 58
- **4.5 Further work** ............................................ 63
- **References** .................................................. 64

## Generalizing the AUGMENT Combinator

**Neil Ghani, Tarmo Uustalu, Varmo Vene**

- **5.1 Introduction** ............................................. 65
- **5.2 Semantics of FOLD/BUILD** ................................. 66
- **5.3 The AUGMENT of free monads** .......................... 69
- **5.4 A generalized AUGMENT combinator** .................. 72
- **5.5 Conclusion and future work** ............................. 77
- **References** .................................................. 77

## Alice Through the Looking Glass

**Andreas Rossberg, Didier Le Botlan, Guido Tack, Th. Brunklaus, Gert Smolka**

- **6.1 Introduction** ............................................. 79
- **6.2 Futures** .................................................. 80
- **6.3 Higher-order modules** ..................................... 83
- **6.4 Packages** ................................................. 84
- **6.5 Components** ............................................. 86
- **6.6 Decomposing components** .................................. 88
- **6.7 Distribution** ............................................. 89
- **6.8 Implementation** ......................................... 92
- **6.9 Related work** ............................................ 92
- **6.10 Outlook** ................................................ 93
- **References** .................................................. 94

## Experiments with GHC’s Optimiser

**László Németh**

- **7.1 Introduction** ............................................. 97
- **7.2 Contributions of the paper** ............................. 98
- **7.3 The setup** ............................................... 98
- **7.4 The method** .............................................. 99
### 8 Disjoint Forms in Graphical User Interfaces

*Sander Evers, Peter Achten, Rinus Plasmeijer*

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Introduction</td>
<td>113</td>
</tr>
<tr>
<td>8.2 FunctionalForms summary</td>
<td>114</td>
</tr>
<tr>
<td>8.3 Combinators for disjoint forms</td>
<td>116</td>
</tr>
<tr>
<td>8.4 Implementation</td>
<td>119</td>
</tr>
<tr>
<td>8.5 Safety</td>
<td>125</td>
</tr>
<tr>
<td>8.6 Related work</td>
<td>126</td>
</tr>
<tr>
<td>8.7 Conclusions and future work</td>
<td>126</td>
</tr>
<tr>
<td>References</td>
<td>127</td>
</tr>
</tbody>
</table>

### 9 A Graphic Functional-Dataflow Language

*Silvia Clerici, Cristina Zoltan*

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Introduction</td>
<td>129</td>
</tr>
<tr>
<td>9.2 Outline of the NiMo language</td>
<td>131</td>
</tr>
<tr>
<td>9.3 The NiMo environment</td>
<td>138</td>
</tr>
<tr>
<td>9.4 Current state and future work</td>
<td>142</td>
</tr>
<tr>
<td>9.5 Concluding remarks</td>
<td>143</td>
</tr>
<tr>
<td>References</td>
<td>144</td>
</tr>
</tbody>
</table>
Preface

The fifth installment of the ‘Trends in Functional Programming’ series was the first one to be organised outside Scotland, the homeland of functional programming that hosted previous installments of TFP and the precursor series, the ‘Scottish Functional Programming Workshops’, going back to 1989. In November 2004, Ludwig-Maximilians University, Munich was the host for TFP04. With the change of venue we had a broad spectrum of submissions both in terms of contents and nationality of the authors: in total 22 papers have been submitted for presentation, and TFP04 itself was attended by 36 participants from 13 countries.

The Symposium on Trends in Functional Programming (TFP) is dedicated to promoting new research directions related to the field of functional programming and to investigate the relationships of functional programming with other branches of computer science. It is designed to be a platform for novel and upcoming research, combined with a post-event refereeing process and a formal publication of selected papers as a book. The presentations submitted this year show a wide spectrum of current research within this scope, from theoretical work on foundations of programming languages to practical usage of functional languages in many different domains.

Reflecting our dedication to promote the work by young researchers in this field, the programme committee for TFP has selected this year for the first time a student paper for the ‘Best Student Paper’ award. We are glad to announce that for TFP04 this award goes to:

Ron van Kesteren, Marko van Eekelen, Maarten de Mol
‘Proof Support for General Type Classes’

We hope that this award will be even more motivation for all young researchers to produce high-quality papers describing their work and to present it in an international forum.

The papers in this volume of TFP cover a wide range of topics from theory, formal aspects of functional programming, graphics and visual programming to distributed computing and compiler design. As is often the case in this community, the most prolific work comes out of the combination of theoretical work with its application on classical problems in functional programming.

The distinguished paper by van Kesteren et al. in Chapter 1 is a good example of this combination, developing proof methods in the SPARKLE system in
order to prove properties common to all instances of a Haskell type class. One fruitful and very active area of functional languages at the moment is generic programming. In Chapter 2, Reig presents a methodology for proving common properties of such generic programs, phrased in type classes within Haskell. The work of Dubois et al. on the FOCAL system in Chapter 3 follows a similar overall direction, integrating proving support into a high-level functional language and supporting advanced language features in the proving component of the integrated system.

Any formal treatment of modern programming languages has to cope with the presence of many different language concepts, and thus concrete formalisations often become unwieldy. In Chapter 4, Hutton and Wright demonstrate how to introduce exceptions into a simple abstract machine by performing equational reasoning on its high-level specification. In Chapter 5, Ghani et al. discuss the category theoretic foundations of combinators such as fold and augment, and extend their results to a wider class of inductive types using these combinators.

The core areas for the TFP series of programming language design and implementation are covered by several papers. In Chapter 6, Rossberg et al. present a functional language for distributed computation, using classical concepts such as futures for synchronisation. In Chapter 7, Németh presents a detailed empirical analysis of the performance of the GHC Haskell compiler, with varying orderings of the optimisation phases, and identifies orderings that consistently improve performance over the default setting for maximal optimisation. In Chapter 8, Evers et al. present a method for the better separation of the logic and the layout of GUI-level forms, and present a library providing an added level of abstraction. Finally, Clerici and Zoltan present in Chapter 9 NiMo, a graphic programming language based on dataflow graphs using lazy evaluation, which provides support for debugging and step-wise program development in its visual programming environment.
Acknowledgements

I would like to thank Martin Hofmann and the other members of the Theoretical Computer Science group of the Department for Informatics at the Ludwig-Maximilians University, Munich, for the help in promoting and organising TFP04. I am indebted to Max Jakob and Spyridon Iliopoulos for valiant on-site help during TFP04. I am particularly grateful to the members of the programme committee and all the referees for a smooth refereeing process. The programme committee for TFP04 was comprised of:

- Stephen Gilmore, University of Edinburgh
- Gaetan Hains, Universite d’Orleans
- Kevin Hammond, University of St Andrews
- John Hughes, Chalmers University
- Hans-Wolfgang Loidl, Ludwig-Maximilians-University Munich (Chair)
- Rita Loogen, Philipps-University Marburg
- Greg Michaelson, Heriot-Watt University Edinburgh
- John O’Donnell, University of Glasgow
- Ricardo Pena, Universidad Complutense de Madrid
- Phil Trinder, Heriot-Watt University Edinburgh
- Marko van Eekelen, University of Nijmegen
- Phil Wadler, University of Edinburgh

Further thanks go to the members of the TFP steering committee for general advice and concrete help in running this event. Last, but not least, I want to thank all participants of TFP04 for making this year’s symposium such a lively event and for making it worth the while for me to organise it.

I am happy to acknowledge financial support by the EU through the APPSEM II Thematic Network and the Mobile Resource Guarantees project, funded as IST-2001-33149 under the European Commission’s Fifth Framework Programme.
Chapter 1

Proof Support for General Type Classes

Ron van Kesteren\textsuperscript{1}, Marko van Eekelen\textsuperscript{1}, Maarten de Mol\textsuperscript{1}

Abstract: We present a proof rule and an effective tactic for proving properties about \textsc{haskell} type classes by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant \textsc{isabelle} handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for more complex forms of overloading. We solve this using an induction scheme derived from the instance definitions. The tactic based on this rule is implemented in the proof assistant \textsc{sparkle}.

1.1 INTRODUCTION

It is often stated that formulating properties about programs increases robustness and safety, especially when formal reasoning is used to prove these properties. Robustness and safety are becoming increasingly important considering the current dependence of society on technology. Research on formal reasoning has spawned many general purpose proof assistants, such as \textsc{coq} \cite{dt04}, \textsc{isabelle} \cite{npw02}, and \textsc{pvs} \cite{osrsc99}. Unfortunately, these general purpose tools are geared towards mathematicians and are hard to use when applied to more practical domains such as actual programming languages.

Because of this, proof assistants have been developed that are geared towards specific programming languages. This allows proofs to be conducted on the source program using specifically designed proof rules. Functional languages are especially suited for formal reasoning because they are referentially transparent.

\textsuperscript{1}Nijmegen Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, Nijmegen, 6525 ED, The Netherlands; Phone: +031 (0)24-3653410; E-mail: rkester@sci.ru.nl, M.vanEekelen@niii.ru.nl, M.deMol@niii.ru.nl
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  x == y = predefinedeqint x y

instance Eq Char where
  x == y = predefinedeqchar x y

instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == [] = False
  [] == (y:ys) = False
  (x:xs) == (y:ys) = x == y && xs == ys

FIGURE 1.1. A type class for equality in HASKELL

1.1.1 Type classes

A feature that is commonly found in functional programming languages is overloading structured by type classes [WB89]. Type classes essentially are groups of types, the class instances, for which certain operations, the class members, are implemented. These implementations are created from the available instance definitions and may be different for each instance. The type of an instance definition is called the instance head. The equality operator will be used as a running example throughout this paper (Figure 1.1).

In the most basic case, type classes have only one parameter and instance heads are flat, that is, a single constructor applied to a list of type variables. Furthermore, no two instance definitions may overlap.

Several significant extensions have been proposed, such as multiple parameters [JIM97], overlapping instances, and instantiation with constructors [Jon93], that have useful applications such as collections, coercion, isomorphisms and mapping. In this paper, the term general type classes is used for systems of type classes that support these extensions and non-flat instance heads. Figure 1.2 shows a multi parameter class for the symmetric operation eq2.

An important observation regarding type classes is that, in general, the defined instances should be semantically related. For example, all instances of the equality operator usually implement an equivalence relation. These properties can be proven for all instances at once by proving them for the available instance definitions. Unfortunately, this is not straightforward because the instance definitions may depend on each other and hence so will the proofs. For example, equality on
Chapter 1. Proof Support for General Type Classes

\begin{verbatim}
class Eq2 a b where
eq2 :: a -> b -> Bool where

instance Eq2 Int Int where
eq2 x y = x == y

instance Eq2 Char Char where
eq2 x y = x == y

instance (Eq2 a b, Eq2 b c) => Eq2 (a, b) [c] where
eq2 (x, y) [u, v] = eq2 x u && eq2 y v

instance (Eq2 a b, Eq2 b c) => Eq2 [c] (a, b) where
eq2 x y = eq2 y x

FIGURE 1.2. A multi parameter class in HASKELL
\end{verbatim}

lists is only symmetric if equality on the list members is so as well.

1.1.2 Contributions

The only proof assistant with special support for overloading that we know of is ISABELLE \cite{Nip93,Wen97}, which essentially supports single parameter type classes and a proof rule for it based on structural induction on types. However, we show that for general type classes, an effective tactic is not easily derived when structural induction is used. We use an induction scheme on types based on the instance definitions to solve this problem. Using this induction scheme, a proof rule and tactic are defined that are both strong enough and effective.

As a proof of concept, we have implemented the tactic in the proof assistant SPARKLE for the programming language CLEAN. The results, however, are generally applicable and can, for example, also be used for HASKELL and ISABELLE, if ISABELLE would support the specification of general type classes. In fact, the examples here are presented using HASKELL syntax. SPARKLE is dedicated to CLEAN, but can also be used to prove properties about HASKELL programs by translating them to CLEAN using the HACLE translator \cite{Nay04}.

1.1.3 Outline

The rest of this paper is structured as follows. First, the proof assistant SPARKLE is presented (Section 1.2). Then, basic definitions for instance definitions, evidence values, and class constrained properties are introduced (Section 1.3). After showing why structural induction does not suffice (Section 1.4), the proof rule and tactic based on the instance definitions are defined (Section 1.5) and extended to multiple class constraints (Section 1.6). We end with a discussion of the imple-
1.2 SPARKLE

The need for this work arose whilst improving the proof support for type classes in SPARKLE. SPARKLE is a proof assistant specifically geared towards CLEAN, which means that it can reason about CLEAN concepts using rules based on CLEAN’s semantics. Properties are specified in a first order predicate logic extended with equality on expressions. An example of this, using a slightly simplified syntax, is:

\[
\text{example: } \forall n:\text{Int}[n \neq \bot] \forall a \forall xs:[a][\text{take } n \text{ xs ++ drop } n \text{ xs } = \text{xs}]
\]

These properties can be proven using tactics, which are user friendly operations that transform a property into a number of logically stronger properties, the proof obligations or goals, that are easier to prove. A tactic is the implementation of (a combination of) theoretically sound proof rules. Whereas in general a proof rule is theoretically simple but not very prover friendly, a tactic is prover friendly but often theoretically more complex. The proof is complete when all remaining proof obligations are trivial. Some useful tactics are, for example, reduction of expressions, induction on expression variables, and rewriting using hypotheses.

In SPARKLE, properties that contain member functions can only be proven for specific instances of that function. For example:

\[
\text{sym}_{\text{Int}}: \forall x:\text{Int}[\forall y:\text{Int}[x == y \rightarrow y == x]]
\]

can be easily proven by induction on lists using symmetry of equality on integers. Proving that something holds for all instances, however, is not possible in general. Consider for example symmetry of equality:

\[
\text{sym}: \forall a[\text{Eq :: } a \Rightarrow \forall x:a \forall y:x[x == y \rightarrow y == x]]
\]

where \(\text{Eq :: } a\) denotes the, previously not available, constraint that equality must be defined for type \(a\). This property can be split into a property for every instance definition, which gives among others the property for the instance for lists:

\[
\text{sym}_{[\text{a}]}: \forall a[\text{Eq :: } a \Rightarrow \forall x:[a] \forall y:[a][x == y \rightarrow y == x]]
\]

It is clear that this property is true as long as it is true for instance \(a\). Unfortunately, this hypothesis is not available. Using an approach based on induction, however, we may be able to assume the hypotheses for all instances the instance definition depends on, and hence will be able to prove the property.

Internally, SPARKLE translates type classes to evidence values or dictionaries [WB89], that make the use of overloading explicit. The evidence value for a class
constraint \( c :: a \) is the evidence that there is an (implementation of the) instance of class \( c \) for type \( a \). Hence, an evidence value exists if and only if the class constraint is satisfied. As usual, we will use the implementation itself as the evidence value. A program is translated by converting all instance definitions to functions (distinct names are created by suffixes). In expressions, the evidence value is substituted for member applications. When functions require certain classes to be defined, the evidence values for these constraints are passed as a parameter. Figure 1.3 shows an example of the result of the translation of the equality class from Figure 1.1.

1.3 PRELIMINARIES

Instead of defining a proof rule that operates on the example properties from Section 1.2, we define both instances and properties at the level that explicitly uses evidence values. In this section, basic definitions for instance definitions, evidence values, and class constrained properties are given.

1.3.1 Instance definitions

Because we intend to support constructor classes, types are formalized by a language of constructors [Jon93]:

\[ \tau ::= \alpha \mid x \mid \tau \tau' \]

where \( \alpha \) and \( x \) range over a given set of type variables and type constructors respectively. For example, \( \tau \) can be \( \text{Int}, [\text{Int}], \) and \( \text{Tree} \ \text{Char} \), but also the \( [], \text{Tree}, \) and \( \rightarrow \) constructors that take types as an argument and yield a list, tree, or function type respectively. \( TV(\tau) \) denotes the set of type variables occurring in \( \tau \). The set of closed types \( \mathcal{T}^c(\tau) \) is the set of types for which \( TV(\tau) \) is empty.

Predicates are used to indicate that an instance of a certain class exists. An instance can be identified by an instantiation of the class parameters. The predicate
\( c :: \bar{\tau} \) denotes that there is an instance of the class \( c \) for instantiation \( \bar{\tau} \) of the class parameters. For example, \( \text{Eq} :: [\text{Int}] \) and \( \text{Eq} :: (\text{Int}, \text{Int}) \) denote that there is an instance of the \( \text{Eq} \) class for types \([\text{Int}]\) and \((\text{Int}, \text{Int})\) respectively:

\[
\pi ::= c :: \bar{\tau}
\]

Because these predicates are used to constrain types to a certain class, they are called \textit{class constraints}. Class constraints in which only type variables occur in the type, for example \( \text{Eq} :: a \), are called \textit{simple}. For reasons of simplicity, it is assumed that all type variables that occur in a class constraint are distinct.

Without loss of generality, throughout this paper we restrict ourselves to type classes that have only one member and no subclasses. Multiple members and subclasses can be supported using records of expressions for the evidence values. An instance definition:

\[
\text{inst} \bar{\pi} \Rightarrow c :: \bar{\tau} = e
\]

defines an instance \( \bar{\tau} \) of class \( c \) for types that satisfy class constraints \( \bar{\pi} \). The instance definition provides the translated expression \( e \) for the class member \( c \). The functions \( \text{Head}(\text{inst} c :: \bar{\pi} \Rightarrow \tau = e) = \tau \) and \( \text{Context}(\text{inst} c :: \bar{\pi} \Rightarrow \tau = e) = \bar{\pi} \) will be used to retrieve the instance head and context respectively.

The program context \( \psi \), that contains the function and class definitions, also includes the available instance definitions. The function \( \text{Idfs}_\psi(c) \) returns the set of instance definitions of class \( c \) defined in program \( \psi \).

### 1.3.2 Evidence values

From the translation from type classes to evidence values, as briefly summarized in Section 1.2, the rule for evidence creation is important for our purpose. Two definitions are required before it can be defined.

Firstly, because instance definitions are allowed to overlap, a mechanism is needed that chooses between them. Since the exact definition is not important for our purpose, we assume that the function \( \text{Ai}_\psi(c :: \bar{\tau}) \) determines the most specific instance definition applicable for instance \( \bar{\tau} \) of class \( c \). \( \text{Ai}_\psi \) is also defined for types that contain variables as long as it can be determined which instance definition should be applied.

Secondly, the \textit{dependencies} of an instance are the instances it depends on:

\[
\text{Deps}(c :: \bar{\tau}, i) = *_{\text{Head}(i) \rightarrow \bar{\tau}}(\text{Context}(i))
\]

where \(*_{\bar{\tau} \rightarrow \bar{\tau}'}\) denotes the substitutor that maps the type variables in \( \bar{\tau} \) such that \(*_{\bar{\tau}}(\bar{\tau}) = \bar{\tau}'\). When \( i \) is not provided, \( \text{Ai}_\psi(c :: \bar{\tau}) \) is assumed for it.

Evidence values are now straightforwardly created by applying the expression of the most specific instance definition to the evidence values of its dependencies:

\[
\text{Deps}(\pi) = (\pi_1, \ldots, \pi_n)
\]
\[
\text{Ai}_\psi(\pi) = \text{inst} c :: \bar{\tau'} \Rightarrow \bar{\tau'} = e
\]
\[
\text{Ev}_\psi(\pi) = \pi_1 \ldots \pi_n
\]
In proofs, evidence values will be created assuming the evidence values for the dependencies are already assigned to expression variables:

\[
\text{Deps}(\pi, i) = \langle \pi_1, \ldots, \pi_n \rangle \\
i = \text{inst } c :: \vec{\pi}' \Rightarrow \vec{\tau} = e \\
\text{Ev}_\psi(\pi, i) = e \ \text{ev}_{\pi_1} \ldots \text{ev}_{\pi_n}
\]

assuming that the evidence for \(\pi\) is assigned to the variable \(\text{ev}_\pi\). A specific instance definition \(i\) can be provided, because \(Ai_\psi(\pi)\) might not be known in proofs.

### 1.3.3 Class constrained properties

In SPARKLE, properties are formalized by a first order predicate logic extended with equality on expressions. The equality on expressions is designed to handle infinite and undefined expressions well.

We extend these properties with class constraints, that can be used to constrain types to a certain class. These properties will be referred to as class constrained properties. For example, consider symmetry and transitivity of equality:

\[
\text{sym: } \forall a [\text{Eq} :: a \Rightarrow \forall x, y : a [\text{ev}_{\text{Eq} :: a} x y \rightarrow \text{ev}_{\text{Eq} :: a} y x]]
\]

\[
\text{trans: } \forall a [\text{Eq} :: a \Rightarrow \forall x, y, z : a [\text{ev}_{\text{Eq} :: a} x y \rightarrow \text{ev}_{\text{Eq} :: a} y z \rightarrow \text{ev}_{\text{Eq} :: a} x z]]
\]

The property \(c :: \vec{\tau} \Rightarrow p\) means that in property \(p\) it is assumed that \(\vec{\tau}\) is an instance of class \(c\) and the evidence value for this class constraint is assigned to \(\text{ev}_{c :: \vec{\tau}}\). Thus, the semantics of the property \(\pi \Rightarrow p\) is defined as \(p[\text{ev}_{\pi} \mapsto \text{Ev}_\psi(\pi)]\).

### 1.4 STRUCTURAL INDUCTION

The approach for proving properties that contain overloaded identifiers taken in ISABELLE essentially is structural induction on types. In this section it is argued that the proof rule for general type classes should use another induction scheme.

Structural induction on types seems an effective approach because it gives more information about the type of an evidence value. This information can be used to expand evidence values. For example, \(\text{ev}_{\text{Eq} :: [\text{a}]}\) can be expanded to \(\text{eqlist}_\text{ev}_{\text{Eq} :: [\text{a}]}\) (see Figure 1.3).

\[
\frac{\forall TV(\pi) [\text{Deps}(\pi) \Rightarrow p(\text{Ev}_\psi(\pi))]}{\forall TV(\pi) [\pi \Rightarrow p(\text{ev}_\pi)]]} \quad \text{(expand)}
\]

More importantly, structural induction allows the property to be assumed for structurally smaller types. Ideally the hypothesis should be assumed for all dependencies on the same class. Unfortunately, structural induction does not always allow this for multi parameter classes.
Consider for example the multi parameter class in Figure 1.2. The instance of \( \text{Eq2} \) for \([\text{Int}] (\text{Char}, \text{Char})\) depends on the instance for \(\text{Char} \text{Int}\), which is not structurally smaller because \(\text{Char}\) is not structurally smaller than \([\text{Int}]\), and \(\text{Int}\) is not structurally smaller than \((\text{Char}, \text{Char})\). Hence, the hypothesis cannot be assumed for this dependency. This problem can be solved by basing the induction scheme on the instance definitions.

1.5 INDUCTION ON INSTANCES

The induction scheme proposed in the previous section can be used on the set of defined instances of a class. In this section, a proof rule and tactic that use this scheme are defined and applied to some examples.

1.5.1 Proof rule and tactic

We first define the set of instances of a class and an order based on the instance definitions on it. The well-founded induction theorem applied to the defined set and order yields the proof rule. Then, the tactic is presented that can be derived from this rule.

Remember that the instances of a class are identified by sequences of closed types. \( \bar{\tau} \) is an instance of class \( c \) if an evidence value can be generated for the class constraint \( c :: \bar{\tau} \). Hence, the set of instances of class \( c \) can be defined as:

\[
\text{Inst}_\psi(c) = \{ \bar{\tau} | \forall c' :: \bar{\tau}' \in \text{Deps}(c :: \bar{\tau}) [\bar{\tau}' \in \text{Inst}_\psi(c')] \}
\]

For example, \( \text{Inst}_\psi(\text{Eq}) = \{ \text{Int}, \text{Char}, [\text{Int}], [\text{Char}], [[\text{Int}}], \ldots \} \).

An order on this set is straightforwardly defined. Because the idea is to base the order on the instance definitions, an instance \( \bar{\tau}' \) is considered one step smaller than \( \bar{\tau} \) if the evidence for \( \bar{\tau} \) depends on the evidence for \( \bar{\tau}' \), that is, if \( c :: \bar{\tau}' \) is a dependency of the most specific instance definition for \( c :: \bar{\tau} \). For example, \( \text{Int} <^1_{(\psi, \text{Eq})} [\text{Int}] \) and \( [\text{Char}] <^1_{(\psi, \text{Eq})} [[\text{Char}}] \).

\[
\bar{\tau} <^1_{(\psi, c)} \bar{\tau}' \iff c :: \bar{\tau}' \in \text{Deps}(c :: \bar{\tau})
\]

Note that there is a specific set of instances for each class and therefore also a specific order for each class.

Well-founded induction requires a well-founded partial order, for which we use the reflexive transitive closure of \( <^1_{(\psi, c)} \). It can be easily derived from the way evidence values are generated that this is indeed a well-founded partial order. Applying this order, \( \leq^*_{(\psi, c)} \), to the well-founded induction theorem yields the following proof rule:

\[
\forall \bar{\tau} \in \text{Inst}_\psi(c) [\forall \bar{\tau}' \leq^*_{(\psi, c)} \bar{\tau} [p(\bar{\tau}') \rightarrow p(\bar{\tau})]] \\
\forall \bar{\alpha} \in \text{Inst}_\psi(c) [p(\bar{\alpha})] \\
\text{(inst-rule)}
\]
Rewriting the proof rule using the definitions of $\text{Inst}_\psi(c)$, $\leq_{(\psi, c)}$, evidence creation, and class constrained properties results in a tactic that can be directly applied to class constrained properties. For all class constraints $c :: \bar{\alpha}$:

$$\forall i \in \text{Idfs}_\psi(c)\forall \text{Head}(i) \in (\forall c)$$

[ $\text{Deps}(c :: \text{Head}(i), i)$

$\Rightarrow \forall c' :: \text{Deps}(c :: \text{Head}(i), i) \left[c = c' \Rightarrow p\left(\text{ev}_{c :: \text{Deps}(c :: \text{Head}(i), i)}, \text{Head}(i)\right)\right]$]

$p\left(\text{Ev}^p\psi(c :: \text{Head}(i), i), \text{Head}(i)\right)$

---

$\forall \bar{\alpha} \in (\forall c)$

where it is assumed that all variables in $\text{Head}(i)$ are fresh. When the tactic is applied to a class constrained property, it generates a proof obligation for every available instance definition with hypotheses for all dependencies on the same class.

1.5.2 Results

The result is both a proof rule and a user friendly tactic. The tactic is nicely illustrated by symmetry of equality (Figure 1.1 and 1.3). When (inst-tactic) is applied to:

**sym:**

$$\forall a \left[\text{Eq} :: a \Rightarrow \forall x :: a \forall y :: a\left[\text{eq}_{\text{Eq} :: a} x y \rightarrow \text{eq}_{\text{Eq} :: a} y x\right]\right]$$

it generates the following three proof obligations (one for each instance definition):

**sym**$_{\text{Int}}$:  

$$\forall x :: \text{Int} \forall y :: \text{Int}\left[\text{eq}_{\text{Int}} x y \rightarrow \text{eq}_{\text{Int}} y x\right]$$

**sym**$_{\text{Char}}$:  

$$\forall x :: \text{Char} \forall y :: \text{Char}\left[\text{eq}_{\text{Char}} x y \rightarrow \text{eq}_{\text{Char}} y x\right]$$

**sym**$_{\text{[a]}}$:  

$$\forall a \left[\text{Eq} :: a \Rightarrow \forall x :: a \forall y :: a\left[\text{eq}_{\text{Eq} :: a} x y \rightarrow \text{eq}_{\text{Eq} :: a} y x\right]\right]$$

which are easily proven using the already available tactics.

The previous step could also have been taken using a tactic based on structural induction on types. However, (inst-tactic) can also assume hypotheses for dependencies that are possibly not structurally smaller. Consider for example the symmetry of $\text{eq}2$ in Figure 1.2:

**sym2:**

$$\forall a, b \left[\text{Eq}2 :: a b \Rightarrow \text{Eq}2 :: b a\Rightarrow \forall x :: a \forall y :: b\left[\text{eq}_{\text{Eq}2 :: a} x y \rightarrow \text{eq}_{\text{Eq}2 :: b} a y x\right]\right]$$
Applying \texttt{(inst-tactic)} to this property generates a proof obligation for every instance definition, including one for the fourth instance of \texttt{Eq2} in Figure 1.2, where \texttt{eq2list} is the translation of that instance definition:

\begin{align*}
\text{sym2}_{[a]} : \forall a, b, c \\
& [ \text{Eq2} :: b \ a \Rightarrow \text{Eq2} :: c \ a] \\
& \Rightarrow [\text{Eq2} :: b \ b \Rightarrow \forall x, y : a[ev_{\text{Eq2} :: b} a x y \Rightarrow ev_{\text{Eq2} :: a} b y x]] \\
& \Rightarrow [\text{Eq2} :: b \ c \Rightarrow \forall x, y : a[ev_{\text{Eq2} :: c} a x y \Rightarrow ev_{\text{Eq2} :: a} c y x]] \\
& \Rightarrow \text{Eq2} :: (b, c) [a] \Rightarrow \forall x : [a] \forall y : [b, c] \\
& \begin{aligned}
& eq2list ev_{\text{Eq2} :: b} a ev_{\text{Eq2} :: c} a x y \\
& \Rightarrow ev_{\text{Eq2} :: (b, c) [a]} y x
\end{aligned}
\end{align*}

In this proof obligation, the hypotheses could not have been assumed when using structural induction on types (see Section 1.4), hence our tactic is useful in more cases.

### 1.6 MULTIPLE CLASS CONSTRAINTS

The proof rule and tactic presented in the previous section work well when the property has only one class constraint. In case of multiple class constraints, however, the rules might not be powerful enough. In this section it is shown that this problem does indeed occur. Therefore, a more general proof rule and tactic are defined and applied to some examples.

#### 1.6.1 The problem

Consider the two class definitions in Figure 1.4. The translated instance definitions are respectively called \texttt{fint}, \texttt{flist}, \texttt{ftree}, \texttt{gint}, \texttt{gtree}, and \texttt{glist} at the level of dictionaries. Given the property:

\begin{align*}
\text{same} : \forall a : [f :: a \Rightarrow g :: a] & \Rightarrow [ev_f :: a x = ev_g :: a x]
\end{align*}

Applying \texttt{(inst-tactic)} yields among others the goal:

\begin{align*}
\text{same}_{[a]} : \forall a : [g :: [a] \Rightarrow \forall x : [a][f \text{list} ev_g :: a x = ev_g :: a x]]
\end{align*}

This goal has a non-simple class constraint, which can only be removed by evidence expansion \texttt{(expand)}, resulting in:

\begin{align*}
\text{same}_{[a]} f' : \forall a : [f :: a \Rightarrow g :: a] & \Rightarrow \forall x : [a][f \text{list} ev_g :: a x \\
& = \text{glist} ev_f :: a ev_g :: a x]
\end{align*}

After some reduction steps, this can be transformed into:
data Tree a = Leaf | Node a (Tree a) (Tree a)

class f a where f :: a -> Bool

instance f Int where
  f x = x == x

instance (g a) => f [a] where
  f [] = True
  f (x:xs) = g x == g x

instance (f a, g a) => f (Tree a) where
  f Leaf = True
  f (Node x l r) = f x == g x

class g a where g :: a -> Bool

instance g Int where
  g x = x == x

instance (f a) => g (Tree a) where
  g Leaf = True
  g (Node x l r) = f x == f x

instance (g a, f a) => g [a] where
  g [] = True
  g (x:xs) = g x == f x

FIGURE 1.4. Problematic class definitions

\[ \forall_a \exists_b \forall_c [ f :: a \Rightarrow g :: a \Rightarrow \forall_{x:a} [ev_{g:a} x = ev_{f:a} x = ev_{g:a} x] ] \]

This proof obligation is true when \( ev_{f:a} x = ev_{g:a} x \). Unfortunately, the induction scheme did not allow us to assume this hypothesis. Since this problem is caused by the fact that the type variables occur in more than one class constraint, the natural solution is to take multiple class constraints into account in the induction scheme.

1.6.2 Proof rule and tactic

We take the same approach as in the previous section. We first define the set of instances, the order, the proof rule and the tactic. Then, in Section 1.6.3, it is shown that the new tactic solves the problem.

First, the set of type sequences that are instances of all classes that occur in a
list of class constraints is defined. \( \bar{\tau} \) is a member of the set if all class constraints \( \bar{\pi} \) are satisfied when all variables \( TV(\bar{\pi}) \) are replaced by the corresponding type from \( \bar{\tau} \). We assume here that \( TV(\bar{\pi}) \) is a linearly ordered, for example lexicographically, sequence and that the elements of \( \bar{\tau} \) are in the corresponding order. For example, \( SetInst(\bar{\pi} :: a, g :: a) = \{ \text{Int}, [\text{Int}], \text{Tree Int}, [[\text{Int}]], \ldots \} \).

\[
SetInst(\bar{\pi}) = \{ \bar{\tau} \mid \forall c :: \bar{\alpha} \in \bar{\pi} = \tau \rightarrow (\bar{\alpha} \in Inst(c)) \}
\]

The order on this set is an extension of the order for single class constraints to sets. A sequence of types \( \tau \) is considered one step smaller than \( \tau' \) if \( *_{TV(\bar{\pi})} \rightarrow_{TV(\bar{\pi})} (\bar{\pi}) \) is a subset of the dependencies of \( *_{TV(\bar{\pi})} \rightarrow_{TV(\bar{\pi})} (\bar{\pi}) \). For example, \( [\text{Int}] \subset \text{Int} \times (\text{Int}) \) and \( [[\text{Int}]] \) because \( \{ \text{f :: [Int], g :: [Int]} \} \) is a subset of \( Deps(g :: [[\text{Int}]] \cup Deps(f :: [[\text{Int}]] \). Here, sequences of class constraints are lifted to sets when required:

\[
\bar{\tau} \subset_{TV(\bar{\pi})} \Rightarrow *_{TV(\bar{\pi})} \rightarrow_{TV(\bar{\pi})} (\bar{\pi}) \subset \bigcup_{\pi \in \bar{\pi}} \text{Deps}(*_{TV(\bar{\pi})} \rightarrow_{TV(\bar{\pi})} (\bar{\pi}))
\]

Again, it can be derived from the evidence creation that the reflexive transitive closure of this order, \( \leq_{TV(\bar{\pi})} \), is a well-founded partial order.

Applying the well-founded induction theorem to this set and order yields the proof rule for multiple class constraints. For every sequence of simple class constraints \( \bar{\pi} \):

\[
\forall \bar{\tau} \in SetInst(\bar{\pi}) \left[ \forall \bar{\tau}' \leq_{TV(\bar{\pi})} \bar{\tau} [p(\bar{\tau}') \rightarrow p(\bar{\tau})] \right] \quad \text{(multi-rule)}
\]

Because multiple class constraints are involved, defining the final tactic is a bit more complicated. Instead of all instance definitions, every combination of instance definitions, one for each class constraint, has to be tried. All of these instance definitions make assumptions about the types of the type variables, and these assumptions should be unifiable. Therefore, we define the most general unifier that takes the sharing of type variables across class constraints into account:

\[
SetMgu((c_1 :: \alpha_1, \ldots, c_n :: \alpha_n), (\tau_1, \ldots, \tau_n)) = * \iff \\
\forall 1 \leq i \leq n[(\bar{\alpha}_i) = \tau_1] \land \forall \bar{\pi} [\forall 1 \leq i \leq n[\bar{\pi} = \tau_i] \Rightarrow \exists \bar{\pi}''[\bar{\pi}' = \bar{\pi}'' \circ \bar{\pi}]]
\]

Furthermore, for readability of the final tactic, some straightforward extensions of existing definitions to vectors are used:

\[
\begin{align*}
Idefs(\pi_1, \ldots, \pi_n) &= \{ i_1, \ldots, i_n \mid i_j \in Idefs(\pi_j) \} \\
Head(i_1, \ldots, i_n) &= \langle \text{Head}(i_1), \ldots, \text{Head}(i_n) \rangle \\
Ev^\psi(\pi_1, \ldots, \pi_n, i_1, \ldots, i_n) &= \langle \text{Ev}^\psi(\pi_1, i_1), \ldots, \text{Ev}^\psi(\pi_n, i_n) \rangle \\
\epsilon v(\pi_1, \ldots, \pi_n) &= \langle \epsilon v(\pi_1), \ldots, \epsilon v(\pi_n) \rangle \\
Deps(\pi_1, \ldots, \pi_n, i_1, \ldots, i_n) &= \langle \text{Deps}(\pi_1, i_1), \ldots, \text{Deps}(\pi_n, i_n) \rangle
\end{align*}
\]

Finally, using the presented definitions, evidence creation, class constrained properties, and the proof rule, the tactic can be defined. For every sequence of simple
class constraints $\bar{\pi}$:

$$
\forall \bar{\pi} \in \text{Idefs}_{\psi}(\bar{\pi}) \exists \bar{\pi}' \in \text{SetMgu}(\bar{\pi}, \text{Head}(\bar{i})) \forall \psi(\text{Head}(\bar{i})) \in \langle T \rangle

\text{[Dep} \bar{\pi}, \bar{i})

\Rightarrow \forall \bar{\pi}' \in \text{Dep}(\bar{\pi}, \bar{i}) \left[ p(\text{ev}_{\bar{\pi}}', \bar{i}', \text{TV}(\bar{\pi})) \right]

\Rightarrow p(\text{Ev}_{\bar{\pi}}(\bar{\pi}, \bar{i}), \bar{i}, \text{Head}(\bar{i}))

\forall \text{TV}(\bar{\pi}) \Rightarrow p(\text{ev}_{\bar{\pi}}, \text{TV}(\bar{\pi})) \quad \text{(multi-tactic)}
$$

Note that applying this tactic may result in non-simple class constraints when non-flat instance types are used. For non-simple class constraints, the induction tactics cannot be applied, but the (expand) rule might be used. However, in practice most instance definitions will have flat types.

This solution for multiple class constraints has some parallels to the constraint set satisfiability problem (CS-SAT), the problem of determining if there are types that satisfy a set of class constraints. The general CS-SAT problem is undecidable. However, recently, an algorithm was proposed [CFV04] that essentially tries to create a type that satisfies all constraints by trying all combinations of instance definitions, as we have been doing in our tactic.

### 1.6.3 Results

In this section, we have generalized the proof rule and tactic from Section 1.5 to multiple class constraints. In case of a single class constraint, the new rules behave exactly the same as (inst-rule) and (inst-tactic). However, now we can apply (multi-tactic) to multiple class constraints at once. Given the previously problematic property:

$$
same: \quad \forall a \quad [f :: a \Rightarrow g :: a \Rightarrow [\text{ev}_f :: a \ x = \text{ev}_g :: a \ x]]
$$

this yields three proof obligations, one for every unifiable combination of instance definitions:

$$
same_{\text{int}}: \quad \forall a \quad [\text{fint} \ x = \text{gint} \ x]
$$

$$
same_{a}: \quad \forall a \quad [f :: a \Rightarrow g :: a \Rightarrow \forall x :: a \text{ [ev}_f :: a \ x = \text{ev}_g :: a \ x]

\Rightarrow \forall x :: a \text{ [flist ev}_f :: a \ x = \text{glist ev}_g :: a \ x]]
$$

$$
same_{\text{Tree} a}: \quad \forall a \quad [f :: a \Rightarrow g :: a \Rightarrow \forall x :: a \text{ [ev}_f :: a \ x = \text{ev}_g :: a \ x]

\Rightarrow \forall x :: \text{Tree} a \text{ [ftree ev}_f :: a \ x = \text{gtree ev}_g :: a \ x]]
$$

The goal same_{a} (and same_{\text{Tree} a}) now has a hypothesis that can be used to prove the goal using the already available tactics. Hence, by taking multiple class constraints into account the problem is solved.
1.7 IMPLEMENTATION

As a proof of concept, we have implemented the (multi-tactic) tactic extended for multiple members and subclasses in SPARKLE. Because of the similarity to the already available induction tactic and the clearness of the code, the implementation of the tactic took very little time. However, to implement the tactic, the typing rules had to be extended. The translation of type classes to dictionaries is only typeable in general using rank-2 polymorphism, which is currently not supported by SPARKLE. This was worked around by handling the dictionary creation and selection in a way that hides the rank-2 polymorphism. Ideally, the use of dictionaries should be completely hidden from the user as well.

The tactic has been used to prove, amongst others, the examples in this paper. The implementation is available at:
http://www.student.kun.nl/ronvankesteren/SparkleGTC.zip

1.8 RELATED AND FUTURE WORK

As mentioned in Section 1.1, the general proof assistant ISABELLE [NPW02] supports overloading and single parameter type classes. ISABELLE’s notion of type classes is somewhat different from HASKELL’s in that it represents types that satisfy certain properties instead of types for which certain values are defined. Nevertheless, the problems to be solved are equivalent. ISABELLE [Nip93, Wen97] uses a proof rule based on structural induction on types, which suffices for the supported type classes. However, if ISABELLE would support more extensions, most importantly multi parameter classes, it would be useful to define our proof rule and a corresponding tactic in ISABELLE.

Essentially, the implementation of the tactic we proposed extends the induction techniques available in SPARKLE. Leonard Lensink proposed and implemented extensions of SPARKLE for induction and co-induction for mutually recursive functions and data types [LvE04]. The main goal was to ease proofs by making the induction scheme match the structure of the program. Together with this work this significantly increases the applicability of SPARKLE.

Because generics is often presented as an extension of type classes [HJ00], it would be nice to extend this work to generics as well. Currently, in CLEAN generics are translated to normal type classes where classes are created for every available data type [AP01]. There is a library for HASKELL that generates classes with boilerplate code for every available data type [LJ03]. The tactic presented here can already be used to prove properties about generic functions by working on these generated type classes. However, the property is only proven for the data types that are used in the program and a separate proof is required for each data type. That is, after all, the main difference between normal type classes and generics. Hence, it remains useful to define a proof rule specifically for generics.
1.9 CONCLUSION

In this paper, we have presented a proof rule for class constrained properties and an effective tactic based on it. Although structural induction on types is theoretically powerful enough, we showed that for an effective tactic an induction scheme should be used that is based on the instance definitions. The tactic is effective, because, using the defined proof rule, it allows all sensible hypotheses to be assumed. The rule and tactic were first defined for single class constraints and then generalized to properties with multiple class constraints.

As a proof of concept, the resulting tactic is implemented in SPARKLE for the programming language CLEAN, but it can also be used for proving properties about HASKELL programs. This is, to our knowledge, the first implementation of an effective tactic for general type classes. If ISABELLE would support extensions for type classes, the tactic could be implemented in ISABELLE as well.

Acknowledgements. We would like to thank Sjaak Smetsers for his suggestions and advice concerning this work, especially on the semantics of type classes in CLEAN, and Fermín Reig for a valuable discussion on generic proofs at the TFP2004 workshop.

REFERENCES


Chapter 2

Generic Proofs for Combinator-based Generic Programs

Fermín Reig

Abstract: Generic programming can bring important benefits to software engineering. In particular, it reduces the burden of verification, since generic proofs can be instantiated at many types. Reasoning about programs that use type classes does not enjoy the benefits of generic reasoning, as it requires providing proofs for an arbitrary number of type instances. This makes the process very impractical. We describe a useful method to reason about a class of programs that use type classes, based on the idea that generic functions implemented using overloading can also be expressed polytypically. We demonstrate the method on functions from the 'scrap-your-boilerplate' library, a collection of combinators for generic programming that has been proposed and implemented recently.

2.1 INTRODUCTION

Generic programming enables concise definitions of functions over many types. In the case of parametric polymorphism, the code of the function is the same at every type. In the case of polytypism, the code is different at each type, but it can be derived mechanically from a single polytypic definition [JJ96, Hin00a].

If similar functions have to be written over and over again for many types, then source code becomes larger. Programming errors increase with program size, and so does the cost of verifying, maintaining and documenting code. Two important benefits of generic programming are smaller programs and smaller proofs.

A number of systems for generic programming have been developed recently [JJ96, Hin00b, AP01, NJ04, LPJ03]. In Generic Haskell [Hin00b], generic func-

1School of Computer Science and I.T., University of Nottingham.
tions are defined by induction on the structure of types. To define a generic function, the programmer provides cases for basic types and for structured types built from sums and products. From this information, a compiler or specialiser can generate instances of the generic function for arbitrary types. Here is an example:

\[
\begin{align*}
\text{add} \langle \text{Bool} \rangle & = (\lor) \\
\text{add} \langle \text{Int} \rangle & = (+) \\
\text{add} \langle A \times B \rangle & (x,y) (x',y') = (\text{add} \langle A \rangle x x', \text{add} \langle B \rangle y y') \\
\text{add} \langle A + B \rangle & (\text{inl } x) (\text{inl } y) = \text{inl } (\text{add} \langle A \rangle x y) \\
\text{add} \langle A + B \rangle & (\text{inr } x) (\text{inr } y) = \text{inr } (\text{add} \langle B \rangle x y) \\
\text{add} \langle A + B \rangle & \_ \_ = \text{error } "\text{shape mismatch}" \\
\end{align*}
\]

This function does point-wise addition of two structures (lists, trees, matrices, . . . ) of the same type and shape. The structures may contain booleans and integers. This style of generic programming has been implemented in Generic Haskell and in Clean \cite{AP01}. Because generic functions take types as parameters in this style, they are also called type-indexed functions. Another proposal, the ‘scrap-your-boilerplate’ library (\cite{LPJ03} \cite{LPJ03, LPJ04}), provides a collection of combinator for traversing values of arbitrary datatypes, together with combinators to simulate dynamic type case. A function similar to the one given above can be written like this:

\[
\text{add } x y = \text{gzipWith } ((\text{mkTT } (+) \ '\text{extTT}' \ (\text{mkTT } (\lor)))) x y
\]

Here, gzipWith pairs matching values in \(x\) and \(y\), and fails if the shapes are different. Then, a dynamic type check on matching values is performed, followed by ‘or’, addition, or failure if the corresponding types are respectively both boolean, both integer, or something else.

The two approaches support different styles of generic programming and are not comparable in terms of the class of generic functions that can be defined.

Little is known about the formal properties of the ‘boilerplate’ combinators. \cite{LPJ03} state the following laws about the combinators \(\text{gmapT}\) and \(\text{gmapQ}\):

\[
\begin{align*}
\text{gmapT } \text{id} & = \text{id} \\
(\text{gmapT } t) \circ (\text{gmapT } t') & = \text{gmapT } (t \circ t') \\
(\text{gmapQ } q) \circ (\text{gmapT } t) & = \text{gmapQ } (q \circ t)
\end{align*}
\]

However, no proofs are provided in that paper. Using the methods proposed by \cite{Pau04} and \cite{vvd05}, it would be possible to prove these laws for a large number of types. Unfortunately, this methodology does not scale too well, since a new proof is required when we consider an additional base type or type constructor. What we really need are generic proofs that cover all possible types.

One of the contributions of this paper is to give generic proofs of the above laws. \cite{LPJ03} use Haskell type classes to define traversal combinators. Our insight is that \(\text{gmapT}\) and \(\text{gmapQ}\) can be defined polytypically instead. We implement them in Generic Haskell, which supports proofs about generic functions (by induction on the structure of the type cases) \cite{Hin00a}. 
A programmer might reasonably expect that similar fusion laws hold for the ‘boilerplate’ combinators that generalise $\text{gmapT}$ and $\text{gmapQ}$. Another contribution of this paper is to show that this is not the case, perhaps contrary to programmers’ intuition. We also calculate conditions that tell us when fusion can be applied to traversals using the more general combinators. This result can be used (by programmers or compilers) to transform code for improved efficiency. To our knowledge, this represents the first published account of formal reasoning about programs that use the ‘boilerplate’ combinators.

### 2.2 ‘BOILERPLATE’ COMBINATORS AS GENERIC FUNCTIONS

The ‘boilerplate’ library is of help when writing programs that do traversals of data built from elaborate, mutually-recursive types. By using a few simple combinators, programmers can avoid writing most of the traversal code by hand. The approach is based on basic combinators that implement one-level, non-recursive traversals. From them, it is possible to implement more complex traversals, such as exhaustive top-down or bottom-up traversal. There are combinators for transforming values and for querying them; and there are purely functional and monadic versions of the combinators.

The basic combinators are $\text{gmapT}$, which applies a generic transformation to the top-level children of a value, and $\text{gmapQ}$, which applies a generic query to the top-level children of a value and returns a list of the results. For instance, let $x$ be the value $(\langle a', (b', c') \rangle)$ and $t$ the transformation that converts characters to upper case and is the identity transformation at any other type. Then $\text{gmapT} t x$ evaluates to $(\langle A', (b', c') \rangle)$. That is, the mapping is not recursive ($x$ has two top-level children, $a'$ and the pair $(b', c')$; $t$ is the identity on pairs). Let $q$ be the query that gets the ASCII number of a character and returns $-1$ for any other type. Then $\text{gmapQ} q x$ evaluates to the list $[97, -1]$.

Because of the regularity of their behaviour, instances of $\text{gmapT}$ and $\text{gmapQ}$ can be generated automatically for every type. Alternatively, $\text{gmapT}$ and $\text{gmapQ}$ can be defined as generic functions. By doing so, we gain the possibility of using existing reasoning principles for generic programs. Here is a polytypic implementation of $\text{gmapT}$. Later in the paper, we will use this version to prove the equivalence $(\text{gmapT} t) \circ (\text{gmapT} t') = \text{gmapT} (t \circ t')$.

```plaintext
\begin{align*}
\text{gmapT}(0) & \quad \_b = b \\
\text{gmapT}(\text{Unit}) & \quad \_u = u \\
\text{gmapT}(\text{Int}) & \quad \_i = i \\
\text{gmapT}(A + B) & \quad t (\text{inl} l) = \text{inl} (\text{gmapT}(A) t l) \\
\text{gmapT}(A + B) & \quad t (\text{inr} r) = \text{inr} (\text{gmapT}(B) t r) \\
\text{gmapT}(A_1 \times \cdots \times A_n) & \quad t (x_1, \ldots, x_n) = (t(A_1) x_1, \ldots, t(A_n) x_n)
\end{align*}
```

Here, 0 is the empty type, whose only value is $\bot$. It is sometimes useful to

---

2 In the current implementation, the combinator $\text{gfoldl}$ is generated, and $\text{gmapT}$ and $\text{gmapQ}$ are defined in terms of it \cite{LPJ04}.
define a datatype with a component of type \( \mathbf{0} \) \cite{Lo04}. In this paper we make use of the case for \( \mathbf{0} \) in some of the proofs. Two comments about this definition of \( \text{gmapT} \) are in order. First, note that \( t \), the first argument to \( \text{gmapT} \), is itself a generic function: it is applied at different types \( (A_1, \ldots, A_n) \) in the last equation. Not all implementations of polytypism support first-class generic functions. Hinze’s recent version, based on type classes \cite{Hin04}, allows them, since generic functions are ordinary Haskell functions. On the other hand, implementations based on translation by specialisation do not support them, although it is the topic of current research \cite{Lo04}. Second, in languages like Generic Haskell, generic functions are defined by providing type cases for binary sums and products, but we have written a case for \( n \)-ary products (using an ellipsis as informal syntax). Supporting \( n \)-ary products in Generic Haskell raises technical difficulties with the types and implementation of generic functions. Fortunately, it is possible to avoid \( n \)-ary products if we translate our definitions to Generic Haskell extended with views \cite{HJL05}. The code for \( \text{gmapT} \) would be almost identical; only the last line would need to be changed to the following:

\[
\text{gmapT} \langle A \times B \rangle t (x, y) = (\text{gmapT} \langle A \rangle t x, \ t \langle B \rangle y)
\]

In this paper, though, we prefer to use \( n \)-ary products, since the resulting definitions correspond more naturally to the English description of the behaviour, namely ‘apply to all top-level children’. If we used the alternative definition using views and binary products, our proofs would require only minor changes, just like the code. Here is our polytypic implementation of \( \text{gmapQ} \).

\[
\begin{align*}
\text{gmapQ} \langle \mathbf{0} \rangle & = [] \\
\text{gmapQ} \langle \text{Unit} \rangle & = [] \\
\text{gmapQ} \langle \text{Int} \rangle & = [] \\
\text{gmapQ} \langle A + B \rangle q (\text{inl } l) & = \text{gmapQ} \langle A \rangle q l \\
\text{gmapQ} \langle A + B \rangle q (\text{inr } r) & = \text{gmapQ} \langle B \rangle q r \\
\text{gmapQ} \langle A_1 \times \cdots \times A_n \rangle q (x_1, \ldots, x_n) & = [\text{q} \langle A_1 \rangle x_1, \ldots, \text{q} \langle A_n \rangle x_n]
\end{align*}
\]

Later in the paper, we will use this definition to prove the equivalence

\[
(\text{gmapQ} \ q) \circ (\text{gmapT} \ t) = \text{gmapQ} \ (q \circ t).
\]

### 2.3 GENERIC PROOFS FOR ‘BOILERPLATE’ COMBINATORS

Generic functions are defined by induction on the structural representation of types. A suitable proof method for such definitions is fixed-point induction \cite{Hin00a}. The main advantage of these proofs is that they are generic, in the sense that a single, type-indexed proof covers all its type instantiations. To construct a proof of

\[^{3}\text{In fact, \cite{Hin04} uses Haskell98. First-class generic functions require rank-2 polymorphism, which is not Haskell98. However, popular Haskell implementations support this extension.}\]
a predicate \( P \) about generic functions, we first write it as a type-indexed predicate \( P\langle T \rangle \). Then, we need to show that it holds for the empty type, for basic types, and that it is preserved by type constructors:

\[
P(0)
\]

\[
P(\text{Unit})
\]

\[
P(\text{Int})
\]

\[
\forall A B.P\langle A \rangle \land P\langle B \rangle \Rightarrow P\langle A + B \rangle
\]

\[
\forall A_1 ... A_n.P\langle A_1 \rangle \land ... \land P\langle A_n \rangle \Rightarrow P\langle A_1 \times \cdots \times A_n \rangle
\]

Fixed-point induction is applicable if \( P \) is an inclusive predicate, i.e. it can be written as a universally quantified conjunction of disjunctions \cite{Sch86}. All the predicates that are proved using fixed-point induction in this paper are inclusive.

### 2.4 A PROOF OF THE FUSION LAW FOR \( gmapT \)

The fusion law for \( gmapT \) says that two consecutive one-level traversals can be fused into one.

**Theorem 2.1.** \( \forall t t' :: (gmapT t) \circ (gmapT t') = gmapT (t \circ t') \)

This law allows a program transformation that reduces the number of traversals over a data structure. We prepare for the proof by stating the fusion law as a type-indexed predicate, with explicit type parameters in applications of generic functions:

\[
P\langle T \rangle = \forall t t' :: (gmapT\langle T \rangle t) \circ (gmapT\langle T \rangle t') = gmapT\langle T \rangle (t \circ t')
\]

We now have to show that \( P\langle T \rangle \) holds for all possible types \( T \). The base case are the empty type and the primitive types.

**• Case** \( T = 0 / \text{Unit} / \text{Int} \)

\[
gmapT{\langle T \rangle} t (gmapT\langle T \rangle t' x)
\]

\[
= \quad \text{(definition of } gmapT\langle 0 / \text{Unit} / \text{Int} \rangle \text{)}
\]

\[
gmapT{\langle T \rangle} t x
\]

\[
= \quad \text{(definition of } gmapT\langle 0 / \text{Unit} / \text{Int} \rangle \text{)}
\]

\[
x
\]

\[
= \quad \text{(definition of } gmapT\langle 0 / \text{Unit} / \text{Int} \rangle \text{)}
\]

\[
gmapT{\langle T \rangle} (t \circ t') x
\]

The induction steps are sums and products. For sums, we need to invoke the induction hypothesis. We only show the case for \text{inl}; the case for \text{inr} is symmetric.

**• Case** \( T = A + B, x = \text{inl } a \)

\[
gmapT{\langle T \rangle} t (gmapT\langle T \rangle t' x)
\]
= \{ \text{definition of } \text{gmap}T\langle A+B \rangle, x = \text{inl} \ a \}\n\text{gmap}T\langle T \rangle \ t \ (\text{inl} \ (\text{gmap}T\langle A \rangle \ t' \ a)) \\
= \{ \text{definition of } \text{gmap}T\langle A+B \rangle \}\n\text{inl} \ (\text{gmap}T\langle A \rangle \ t \ (\text{gmap}T\langle A \rangle \ t' \ a)) \\
= \{ \text{induction} \}\n\text{inl} \ ((\text{gmap}T\langle A \rangle \ t \circ t') \ a) \\
= \{ \text{definition of } \text{gmap}T\langle A+B \rangle, x = \text{inl} \ a \}\n\text{gmap}T\langle T \rangle \ (t \circ t') \ x

For tuples, we don’t even need the induction hypothesis; we only make use of the definitions of \text{gmap}T and the function composition operator.

- **Case** \( T = A_1 \times \cdots \times A_n, x = (x_1, \ldots, x_n) \)

\[
\begin{align*}
\text{gmap}T\langle T \rangle \ t \ (\text{gmap}T\langle T \rangle \ t' \ x) \\
= \{ \text{definition of } \text{gmap}T\langle A_1 \times \cdots \times A_n \rangle \}\n\text{gmap}T\langle T \rangle \ t \ (t'\langle A_1 \rangle \ x_1, \ldots, t'\langle A_n \rangle \ x_n) \\
= \{ \text{definition of } \text{gmap}T\langle A_1 \times \cdots \times A_n \rangle \ ; \text{function composition} \}\n((t \circ t')\langle A_1 \rangle \ x_1, \ldots, (t \circ t')\langle A_n \rangle \ x_n) \\
= \{ \text{definition of } \text{gmap}T\langle A_1 \times \cdots \times A_n \rangle \}\n\text{gmap}T\langle T \rangle \ (t \circ t') \ x
\end{align*}
\]

The other fusion theorem says that a one-level traversal followed by a one-level query can be fused into a one-level query.

**Theorem 2.2.** \( \forall t :: (\text{gmap}Q \ q) \circ (\text{gmap}T \ t) = \text{gmap}Q \ (q \circ t) \)

The proof proceeds in a similar way to the one shown above. It is included in the full version of the paper [Rei05]. We omit the proof of \text{gmap}T \ id = id, since it is similar to these two.

In the next section, we give a function over lists and a theorem about that function. We then generalise to a generic function over arbitrary types, and a corresponding theorem. Finally, we construct a generic proof of the theorem.

### 2.5 A THEOREM ABOUT \textit{occurs}

Consider the following polymorphic function that counts the number of occurrences of a value in a list:

\[
\begin{align*}
\text{occursList} & \in (\text{Eq} \ a) \Rightarrow a \rightarrow [a] \rightarrow \text{Int} \\
\text{occursList} \ v & = \text{length} \circ \text{filter} \ (\equiv \ v)
\end{align*}
\]

This performs two list traversals. A version that makes only one is easy to write, but the one shown here is slightly more concise; consider it a specification, but one that is executable. This function satisfies the following theorem:
Theorem 2.3. \( \forall x \; xs \; f :: \text{occursList} \; x \; xs \leq \text{occursList} \; (f \; x) \; (\text{map} \; f \; xs) \) 

Similar functions for other data structures, such as trees, are likely to be useful as part of a complete library. For each of them we would also state a similar theorem. Even better is to write a single generic function, state a single theorem, and write one proof only. Here is such a generic function, implemented with the ‘boilerplate’ combinators.

\[
\text{occurs} \; x \; xs = \text{everything} \; (+) \; (0 \cdot \text{mkQ} \; \text{cnt}) \; xs
\]

where
\[
\text{cnt} \; y = \text{if } y \equiv x \text{ then } 1 \text{ else } 0
\]

This traverses every node of \( xs \), produces an integer for each one of them, and adds them all together. We describe the code briefly, but refer the reader to [LPJ03] for the details of the combinators \text{everything} and \text{mkQ}. The combinator \text{mkQ} turns functions into generic queries that can be applied to arguments of any type. If a node \( n \) has the same type as \( x \), then \( \text{cnt} \) is applied to \( n \); otherwise, the default value 0 is returned. The function \text{everything} is defined as follows: \text{everything} \; k \; q \; x = \text{foldl} \; k \; (q \; x) \; (\text{gmapQ} \; (\text{everything} \; k \; q) \; x)

A generic theorem

The theorem that the generic function satisfies is the following:

Theorem 2.4.

\[
\forall x \; xs \; f :: \text{fusable}(f) \Rightarrow \text{occurs} \; x \; xs \leq \text{occurs} \; (f \; x) \; (\text{everywhere} \; (\text{mkT} \; f) \; xs)
\]

The function \text{everywhere} traverses values bottom-up. Here, it applies the generic transformation \( (\text{mkT} \; f) \) to every node of \( xs \), bottom-up. Its definition is:

\[
\text{everywhere} \; t = t \circ \text{gmapT} \; (\text{everywhere} \; t)
\]

The combinator \text{mkT} makes a generic transformation from a function, that is, it extends the function to arguments of any type. Its specification is:

\[
(\text{mkT} \; f) \; x = f \; x, \text{ if the type of } x \text{ is the same as the domain of } f
\]

\[
(\text{mkT} \; f) \; x = x, \text{ otherwise}
\]

The predicate \text{fusable} says that an \text{everywhere} transformation followed by an \text{everything} query, can be performed as a single traversal:

\[
\text{fusable}(f) = \forall x \; q :: \text{everything} \; (+) \; q \; (\text{everywhere} \; (\text{mkT} \; f) \; xs) = \text{everything} \; (+) \; (q \circ (\text{mkT} \; f)) \; xs
\]

This generalises Theorem 2.2 from one-level traversals to full bottom-up traversals. We believe (but have not proved yet) that this predicate is satisfied for transformations that do not alter the shape of the value \( xs \).
The reader may wonder why the generic theorem includes this condition, making it weaker than the specific theorem about lists. The ‘boilerplate’ implementation of \textit{occurs} is more general and can be used to count occurrences of a value \(x\) of type \(a\), not only in a container of \(as\), but in a structure of arbitrary type. In general, an \textit{everywhere} transformation is more general than \textit{map} over a container type. For instance, the transformation can remove elements from a container, which \textit{map} is guaranteed not to do. These more general functions satisfy weaker properties.

\textbf{A generic proof}

Our first attempt at a generic proof resulted in a long, complex proof by induction on types. Just like large programs, long proofs written by hand are more likely to contain errors. Subsequently, we managed to produce a concise proof where only a few lemmas require proofs by induction on types. In the proof, we use a version of the code where the function \textit{cnt} has been lambda-lifted because having \(x\) as a free variable in \textit{cnt} complicates things.

\[
\text{occurs } x \; xs = \text{everything } (+) \; (q \; x) \; xs \quad \text{where} \\
q \; x = (0 \text{ `mkQ` } (cnt \; x)) \\
cnt \; x = \lambda y \rightarrow \text{if } x \equiv y \text{ then } 1 \text{ else } 0
\]

We make use of a number of lemmas. The first two say that \textit{gmapQ} and \textit{everything} preserve ordering.

\textbf{Lemma 2.1.} \(\forall q \; q' :: q \leq q' \Rightarrow \text{gmapQ } q \leq \text{gmapQ } q'\)

\textbf{Lemma 2.2.} \(\forall q \; q' :: q \leq q' \Rightarrow \text{everything } (+) \; q \leq \text{everything } (+) \; q'\)

We use the symbol \(\leq\) to denote point-wise ordering between functions: \(f \leq g = \langle \forall x :: f \; x \leq g \; x \rangle\). When \(xs\) and \(ys\) are lists, we say that \(xs \leq ys\) if they have the same length and corresponding elements are related by \(\leq\).

For conciseness, we define the following shorthand. \(q' \; f \; x = (q \; (f \; x)) \circ (mkTf)\), where \(q\) is the query from the \textbf{where} clause in the definition of \textit{occurs}. It is used in our last lemma.

\textbf{Lemma 2.3.} \(\forall x \; f :: q \; x \leq q' \; f \; x\)

Proofs of the lemmas are included in an appendix. Finally, with the help of all this, we can give an equational proof of \textbf{Theorem 2.4} that is simple and short.
\( \text{occurs } x \; xs \)
\[= \{ \text{definition of } \text{occurs} \} \]
\( \text{everything } (+) \; (q \; x) \; xs \)
\[= \{ \text{definition of } \text{everything} \} \]
\( \text{foldl} \; (+) \; (q \; x \; xs) \; (\text{gmapQ} \; (\text{everything} \; (+) \; (q \; x)) \; xs) \)
\[\leq \{ \text{Lemmas 2.1, 2.2, 2.3; definitions of } \text{foldl} \; \text{and } \leq \} \]
\( \text{foldl} \; (+) \; (q' \; f \; x \; xs) \; (\text{gmapQ} \; (\text{everything} \; (+) \; (q' \; f \; x)) \; xs) \)
\[= \{ \text{definition of } \text{everything} \} \]
\( \text{everything} \; (+) \; (q' \; f \; x) \; xs \)
\[= \{ \text{definition of } q' \} \]
\( \text{everything} \; (+) \; ((q \; (f \; x)) \circ (\text{mkT} \; f)) \; xs \)
\[= \{ \text{condition } (\text{fusable } (f)) \} \]
\( \text{everything} \; (+) \; (q \; (f \; x)) \; (\text{everywhere } (\text{mkT} \; f) \; xs) \)
\[= \{ \text{definition of } \text{occurs} \} \]
\( \text{occurs} \; (f \; x) \; (\text{everywhere } (\text{mkT} \; f) \; xs) \)

### 2.6 A FUSION LAW FOR everywhere

In the previous sections we have used generic reasoning to prove theorems about
generic functions. Here we use it to calculate the weakest condition needed for a
statement to hold. The statement in question is a generalisation of Theorem 2.1
from a one-layer transformation to a full bottom-up transformation.

**Statement.** \(<\forall t \; t' :: (\text{everywhere } t) \circ (\text{everywhere } t') = \text{everywhere } (t \circ t')>\)

As with other fusion laws, its practical importance is that it can be used to op-
timise programs. When this is done reliably and consistently, programmers are
couraged to write simple functions, and let the compiler turn them into efficient
ones. (Like our \text{occursList} function in § 2.5.)

Alas, it is not the case that two arbitrary \text{everywhere} transformations can be
fused. For instance, let \(t\) be a transformation that prepends a fixed character,
say ‘a’, to a string and is the identity at other types. It can be defined as
t = \text{mkT} \; (\lambda s \rightarrow 'a' : s) . Then we have that \text{everywhere} \; (t \circ t) applied to the empty
string returns "aa", while \text{everywhere} \; (t \circ t) returns "aaa" \footnote{The first application of \text{everywhere } t \ returns 'a' : []. If 'a' is prepended,
bottom-up, to all nodes of type string in 'a' : [], we get "aaa".}

What happens here is that the first transformation creates new nodes where
the second transformation is applicable – in our case, new substrings. That is, it
changes the structure. A (too strong) condition for fusion is that the first traversal does not change the structure. A weaker condition is that at any nodes that are created or deleted, the second transformation is the identity. What we do in this section is turn this informal description into a precise statement about $t$ and $t'$, via formal calculation. In fact, it is possible that the informal condition is not the weakest one – this is the problem with informal statements. However, the one we calculate is guaranteed to be, by construction, the weakest condition. This is the case because all steps in the proof are equality steps.

This calculation is more interesting than the proofs we have shown so far, in the sense that straightforward induction on the structure of types is not enough to complete the proof.

### 2.6.1 Calculating a condition for fusion

We proceed as if we were writing a proof, even though we know the statement is not a theorem. Then, when a step cannot be justified, we make it a condition of the statement. Our theorem can be stated as:

\[
\forall t \ t' :: (\text{everywhere } t) \circ (\text{everywhere } t') = \text{everywhere } (t \circ t') \iff \text{fusable}_{\text{ev}} (t, t')
\]

The predicate $\text{fusable}_{\text{ev}} (t, t')$ is the conjunction of the conditions that arise during the calculation. We defer giving its definition until we complete the proof. Conditions only arise in the cases for sums and products. The statement is already provable for the empty type and for basic types:

- **Case $T = 0 / \text{Unit} / \text{Int}$**

\[
\text{everywhere}(T) t \ (\text{everywhere}(T) t' x)
= \{ \text{definitions of everywhere and } \text{gmapT}(0 / \text{Unit} / \text{Int}) \ \}
\]

\[
\text{everywhere}(T) t \ (t' x)
= \{ \text{definitions of everywhere and } \text{gmapT}(0 / \text{Unit} / \text{Int}) \ \}
\]

\[
(t \circ t')(T) x
= \{ \text{definitions of everywhere and } \text{gmapT}(0 / \text{Unit} / \text{Int}) \ \}
\]

\[
\text{everywhere}(T) \ (t \circ t') x
\]

The case for sums shows that straightforward induction is not sufficient to complete the proof. Assume that we used the following induction hypothesis:

\[
I(T) = (\forall t \ t' :: \text{everywhere}(T) t \circ \text{everywhere}(T) t' = \text{everywhere}(T) (t \circ t'))
\]

In the third equality step, in order to proceed towards the goal, we would like to use the fusion law (backwards) to split the traversal $(\text{everywhere } (t \circ t'))$ into a composition of two traversals $(\text{everywhere } t \circ \text{everywhere } t')$. But we may not invoke the induction hypothesis as a justification, since, there, it is $\text{gmapT}$ that takes
a type argument, and the induction hypothesis is about *everywhere*, not *gmapT*. The solution is to strengthen the induction hypothesis and construct a proof by mutual induction. Thus, the induction hypothesis is:

\[
I'(T) = \langle \forall t' :: I(T) \land \\
gmap T(T) (\text{everywhere } t \circ \text{everywhere } t') = gmap T(T) (\text{everywhere } (t \circ t')) \rangle
\]

In the remainder of the proof we sometimes use *evW* instead of *everywhere* in order to limit the size of a line to the page width.

- **Case** \( T = A + B, x = \text{inl } a \)

\[
\text{everywhere}(T) (t \circ t') x \\
= \{ \text{definition of } everywhere \} \\
(t \circ t')(T) \ (gmap T(T) (\text{everywhere } (t \circ t')) x) \\
= \{ \text{definition of } gmap T(A + B) \} \\
(t \circ t')(T) \ (\text{inl } (gmap T(A) (\text{everywhere } (t \circ t')) a)) \\
= \{ \text{(mutual) induction} \} \\
(t \circ t')(T) \ (\text{inl } (gmap T(A) (\text{everywhere } (t \circ t')) a)) \\
= \{ \text{definition of } gmap T(A + B) \} \\
(t \circ t')(T) \ (gmap T(T) (\text{everywhere } (t \circ t')) x) \\
= \{ \text{gmapT fusion} \} \\
(t \circ t')(T) \ (((gmap T(T) (\text{evW } t)) \circ (gmap T(T) (\text{evW } t'))) x) \\
= \{ \text{• condition} \} \\
t(T) ((gmap T(T) (\text{evW } t)) \circ t(T) \circ (gmap T(T) (\text{evW } t'))) x \\
= \{ \text{definition of } everywhere, \text{ twice} \} \\
\text{everywhere}(T) t (\text{everywhere}(T) t' x)
\]

The case for \( x = \text{inr } b \) is symmetric.

- **Case** \( T = A_1 \times \cdots \times A_n, x = (x_1, \ldots, x_n) \)

\[
\text{everywhere}(T) t (\text{everywhere}(T) t' x) \\
= \{ \text{definition of } everywhere \} \\
\text{everywhere}(T) t (t'(T) (gmap T(T) (\text{everywhere } t') x)) \\
= \{ \text{definition of } everywhere \} \\
t(T) (gmap T(T) (\text{everywhere } t) (t'(T) (gmap T(T) (\text{everywhere } t') x))) \\
= \{ \text{• condition} \} \\
(t \circ t')(T) ((gmap T(T) (\text{everywhere } t) \circ gmap T(T) (\text{everywhere } t'))) x \\
= \{ \text{gmapT fusion} \} \\
(t \circ t')(T) ((gmap T(T) (\text{everywhere } t \circ \text{everywhere } t'))) x \\
= \{ \text{definition of } gmap T(A_1 \times \cdots \times A_n) \} \\
\]
\[(t \circ t')\langle T \rangle \left( (evW t \circ evW t')\langle T_1 \rangle x_1, \ldots, (evW t \circ evW t')\langle T_n \rangle x_n \right) \]
\[= \{ \text{induction} \} \]
\[= \{ \text{definition of } gmapT\langle A_1 \times \cdots \times A_n \rangle \} \]
\[= \{ \text{definition of } everywhere \} \]
\[= everywhere\langle T \rangle (t \circ t') x \]

For both sums and products we get the same condition. It remains that we prove the second conjunct of the induction hypothesis. This part of the proof is in Appendix 2.11.

To summarise, we have replaced an informal condition on \(t\) and \(t'\) ("everywhere \(t\) preserves the shape, except possibly at nodes where \(t'\) is the identity"), by a precise predicate, defined in terms of \(gmapT\) and \(everywhere\).

\[fusable_{ev} (t, t') = t \circ gmapT (everywhere t) \circ t' \circ gmapT (everywhere t') = t \circ t' \circ gmapT (everywhere t) \circ gmapT (everywhere t')\]

This predicate, however, is rather complex, and it is not easy to check whether it holds in the context of specific datatypes and for specific transformations \(t\) and \(t'\). In practise, it might be easier to check the following, stronger predicate.

\[fusable_{ev'} (t, t') = gmapT (everywhere t) \circ t' = t' \circ gmapT (everywhere t)\]

We still do not have experience in using these conditions in real-world situations. Only time will tell how effective they are in practise.

2.7 CONCLUSIONS AND FURTHER WORK

The essence of this paper is the idea that reasoning about (a restricted class of) overloaded functions can be practical, provided that those functions can be described in a generic programming style that supports adequate proof methods. When such a translation is not possible (because the function does not admit a generic definition), reasoning is not impossible, but instead of a single generic proof, proofs for many type instances must be provided [vvd05, Pau04].

We have looked at functions from the ‘boilerplate’ library, but the principle is general and is applicable to overloaded functions that can be generated mechanically (e.g. using Haskell’s deriving clauses, or with tools like DrIFT\(^5\)).

Our conviction that proofs are important for generic programs has been strengthened as a result of writing this paper. When we first set out to prove theorems 2.4 and 2.5, we believed that stronger versions of those theorems were true. After all,

\(^5\)Available from [http://repetae.net/john/computer/haskell/DrIFT](http://repetae.net/john/computer/haskell/DrIFT)
they looked like straightforward generalisations of known theorems about functions over lists. The confusion can arise because in many cases, \textit{everywhere} computes the same result as a generic \textit{map}. However, \textit{everywhere} is a more general function than a generic \textit{map} – for instance, its application can result in a change of the structure of a container – and satisfies fewer properties. The danger of this sort of mistake is that programmers might perform program transformations (perhaps to improve performance) that are not meaning-preserving.

An interesting area for further work is the application of reasoning to program transformation. A practical disadvantage of generic functions implemented with the ‘boilerplate’ combinators is the performance penalty caused by dynamic type checks. When enough is known about the types of arguments, it is sometimes possible to replace calls to generic functions by calls to specialised functions. For instance, assume that \( f \) has type \( \text{Int} \rightarrow \text{Int} \). Then, it is easy to prove that \( \text{everywhere} \ (\text{mkT} \ f) \ xs = \text{map} \ f \ xs \) if \( xs \) is a list of \text{Ints}. We also have that \( \text{everywhere} \ (\text{mkT} \ f) \ ys = ys \) if \( ys \) is a list of characters, or in general, a value whose type does not contain \text{Ints}. These transformations eliminate the costs of dynamic type checks and could be performed automatically by a compiler. Being able to reason about generic functions is essential in order to justify the soundness of such transformations.

\textbf{Acknowledgements.} I would like to thank Roland Backhouse for very helpful discussions and feedback. Ralf Lämmel pointed out several errors and suggested improvements to a previous version of the paper. I would also like to thank the anonymous reviewers for their feedback. This work is fundend by EPSRC grant GR/S27078/01.

\section{2.8 Proof of Lemma 2.1}

\( \langle \forall q q' :: q \leq q' \Rightarrow \text{gmapQ} \ q \leq \text{gmapQ} \ q' \rangle \)

By fixed-point induction on the structure of the type argument. The proof is included in the full version of the paper [Rei05].

\section{2.9 Proof of Lemma 2.2}

\( \langle \forall q q' :: q \leq q' \Rightarrow \text{everything} \ (+) q \leq \text{everything} \ (+) q' \rangle \)

By fixed-point induction on the structure of the type argument. The proof is by mutual induction using the following induction hypothesis:

\( \langle \forall q q' :: q \leq q' \Rightarrow (\text{everything} \ (+) q \leq \text{everything} \ (+) q' \land \text{gmapQ} \ (\text{everything} \ (+) q) \leq \text{gmapQ} \ (\text{everything} \ (+) q')) \rangle \)
2.10 PROOF OF LEMMA 2.3

\[ \langle \forall x : q x \preceq q' f x \rangle \]

We prove the statement \( \langle \forall f x : q x \preceq q' f x \rangle \). Lemma 2.3 follows from this statement and the definition of \( \preceq \).

- **Case** \( x \) and \( xs \) have different type.

  \[
  q \ x \ xs \\
  = \quad \{ \text{\( x \), \( xs \) have different type; definition of \( q \) } \} \\
  0 \\
  = \quad \{ (f \ x), ((mkT f) \ xs) \text{ have different type; definition of \( mkQ \) } \} \\
  (0 \ 'mkQ' \ (f \ x)) \ ((mkT f) \ xs) \\
  = \quad \{ \text{definition of \( q' \) } \} \\
  q' f \ x \ xs
  \]

- **Case** \( x \) and \( xs \) have the same type.

  We distinguish two cases. When \( x \neq xs \), then \( x \) 'occurs' 0 times in \( xs \).

  \[
  q \ x \ xs \\
  = \quad \{ x \neq xs; \text{definition of } q \} \\
  0 \\
  \leq \quad \{ \text{definition of } q' \} \\
  q' f \ x \ xs
  \]

  If \( x = xs \), then \( x \) 'occurs' once in \( xs \), and both \( q \) and \( q' f \) return 1.

  \[
  q \ x \ xs \\
  = \quad \{ x = xs; \text{definition of } q \} \\
  1 \\
  = \quad \{ \text{definition of } q \} \\
  q (f \ x) \ (f \ x) \\
  = \quad \{ (f \ x) \text{ well typed; definition of } mkT \} \\
  q (f \ x) \ ((mkT f) \ x) \\
  = \quad \{ x = xs; \text{definition of } q' \} \\
  q' f \ x \ xs
  \]

2.11 PROOF OF THEOREM 2.5 (CONTINUED)

Below is the proof, by fixed-point induction, of the second conjunct of the induction hypothesis:
Chapter 2. Generic Proofs for Combinator-based Generic Programs

\(\forall t\ t' :: \text{everywhere } t \circ \text{everywhere } t' = \text{everywhere } (t \circ t') \land \ gmapT(\text{everywhere } t \circ \text{everywhere } t') = gmapT(\text{everywhere } (t \circ t'))\)

- **Case** \(T = 0 / \text{Unit} / \text{Int}\)

\[
gmapT(T) (\text{everywhere } (t \circ t')) x
= \{ \text{definition of } gmapT(0 / \text{Unit} / \text{Int}) \}
x
= \{ \text{definition of } gmapT(0 / \text{Unit} / \text{Int}) \}
gmapT(T) (\text{everywhere } t \circ \text{everywhere } t') x
\]

- **Case** \(T = A + B, x = \text{inl } a\)

\[
gmapT(T) (\text{everywhere } (t \circ t')) x
= \{ \text{definition of } gmapT(A + B) \}
\text{inl} (gmapT(A) (\text{everywhere } (t \circ t')) a)
= \{ \text{induction} \}
\text{inl} (gmapT(A) (\text{everywhere } t \circ \text{everywhere } t') a)
= \{ \text{definition of } gmapT(A + B) \}
gmapT(T) (\text{everywhere } t \circ \text{everywhere } t') x
\]

- **Case** \(T = A + B, x = \text{inr } b\)

Similar

- **Case** \(T = A_1 \times \cdots \times A_n, x = (x_1, \ldots, x_n)\)

\[
gmapT(T) (\text{everywhere } t \circ \text{everywhere } t') x
= \{ \text{definition of } gmapT(A_1 \times \cdots \times A_n) \}
((\evW t \circ \evW t') (T_1) x_1, \ldots, (\evW t \circ \evW t') (T_n) x_n)
= \{ \text{(mutual) induction} \}
(\text{everywhere}(T_1) (t \circ t') x_1, \ldots, \text{everywhere}(T_n) (t \circ t') x_n)
= \{ \text{definition of } gmapT(A_1 \times \cdots \times A_n) \}
gmapT(T) (\text{everywhere } (t \circ t')) x
\]

**REFERENCES**


Chapter 3

Building certified components within FOCAL

Catherine Dubois\textsuperscript{1}, Thérèse Hardin\textsuperscript{2}, Véronique Viguié Donzeau Gouge\textsuperscript{3}

Abstract: Existing provers are good to prove program properties but bad in language support to write large programs. This combination is tackled in the program development environment FOCAL, a framework dedicated to the complete development of certified components, from the specification stage to the implementation one. FOCAL incorporates features such as inheritance, abstraction, late binding and redefinition. The paper gives an overview of the design of the environment and presents the language itself and the associated tools.

3.1 INTRODUCTION

Software developed for safety or security critical systems (aircrafts, automatic public transportation, nuclear plants, credit cards etc.) must pass a high level of scrutiny to ensure that it conforms to standards such as the IEC61508 standards for hard/soft systems, at all stages of development from requirements analysis through documentation and testing. For high levels of criticality, these standards require to use not only methods like safety analysis and careful code reviewing, but also formal methods along the complete software life-cycle process, in order to demonstrate correctness, safety, reliability, and security properties. There is a urgent need for tools able to help engineers along that process to decrease development and evaluation complexity and costs. Indeed static analysers, provers,

\textsuperscript{1}CEDRIC, IIE, 18 allée Jean Rostand, 91025 Evry FRANCE; E-mail: dubois@iie.cnam.fr
\textsuperscript{2}CEDRIC, CNAM, 292 rue Saint Martin, 75003 Paris FRANCE; E-mail: donzeau@cnam.fr
\textsuperscript{3}LIP6, UPMC, 8 rue du Capitaine Scott, 75015 Paris FRANCE; E-mail: therese.hardin@lip6.fr
verifiers are great help to deal with correctness but they are definitively not usable to write large programs. This problem is tackled here. And we claim that functional languages and, more generally an approach based on functionality, may lead to powerful development environments, able to handle most of the safety requirements.

Since 1998 we address the problem of highly critical software development, and we have started to build a new development environment, called FOCAL (previously called FOC) freely distributed at [focal.inria.fr](http://focal.inria.fr). Its language is a strict functional ML-like language, with declarations, definitions of functions, properties and proofs. It comes with a compiler which produces runnable code, automatic documentation facilities and two ways to prove properties automatically or not, the proofs being ultimately checked with the theorem prover Coq. Presently, it provides no help for treating concurrency, testing, model-checking, all points which have to be integrated to reach our objectives. Thus it is still in an early stage of development but we think that it already deserves to be presented, as it can help specification, design and implementation steps of critical components. Moreover, its functional style and its modularity provide help for ultimate code reviewing. We are currently integrating tools based on first-order rewriting and equational theories.

In the following, we first give the requirements of FOCAL and its fundamental design principles in the Section 3.2. We give an overview of the language itself in the Section 3.3. Then, Section 3.4 details a complete, but short, example. Section 3.5 presents the compiler, the proof tools and the documentation tools. Some applications developed within FOCAL are briefly presented in 3.6. A very short comparison with other approaches is given in Section 3.7. The conclusion presents some development perspectives.

### 3.2 FOCAL REQUIREMENTS AND DESIGN PRINCIPLES

The project of developing software meeting high level safety requirements started in 1998, after some unsuccessful attempts [Ale98, BHMMR99, SBR99] of proving properties of computer algebra programs. The main difficulties came from the ambiguities of the semantics of the language. We refused to choose between several possible interpretations of, for example, multiple inheritance and late binding, as we cannot have the guarantee that compilers conform to our choice. The FOCAL project arises from this conclusion with the objective of helping all stages of development of critical software within safety and security framework, at least when formal methods are required. In this section, we briefly expose the requirements for the development environment and relate them to the fundamental design principles of FOCAL.

From the beginning of our project, we have a double requirement: the development environment must provide high-level and justified confidence to users and it must be easy to use by well-trained engineers. Thus, some balance has to be found between the logical and semantic rigour and the cost of formal development, partly due to the lack of training of engineers on logics and provers.
First of all, we consider that a specification is defined by declarations and properties, and also by some comments in natural language which are true parts of the documentation. The design phase may introduce more declarations, some definitions of functions, data representation and related properties. At this stage functions represent effective relations between even not yet implemented entities. The implementation phase adds more definitions and properties until the implementation is complete, that is, all declarations have received definitions and all properties are proven. Actually, proofs can be done ‘just in time’, a point explained further.

So the user of the environment develops libraries of components, going step by step from specifications to programs actually reusing already designed or implemented components. This includes proving the required properties. Thus, the language has to provide declarations with deferred definitions and redefinitions, multiple inheritance, powerful parametrization, sharing/reusing of code etc. and definitively, a powerful type system. It also has to provide a user-friendly prover or at least a nice interaction with an existing prover.

The components of the library can also be used as Components Off The Shelf (COTS) in a kind of read-eval-print loop, having confidence in imported components. These end users do not want to do proofs but have to be protected from misuse of components. This last point asks for an abstraction mechanism over components.

To sum up, we want to address certified development in the large but also certified development in the small. So FOCAL proposes a language featuring mechanisms such as inheritance, late binding, redefinition, parametrization, encapsulation and declarations/definitions, properties/theorems. The confidence in proofs relies on proof verification.

### 3.3 OVERVIEW OF FOCAL

In this section we introduce the main features of the language in its current version. The way proofs can be done will be described in Section 3.5. A complete example is given in the next section. A FOCAL development is organised as a hierarchy which may have several roots. The upper levels of the hierarchy are built along the specification stage while the lower ones correspond to implementations. Each node of the hierarchy corresponds to a progress to a complete implementation. We call refinement the process of building top-down a hierarchy.

#### Species

A node of the hierarchy is called a species. A species is a kind of record composed of fields, called methods, which may be:

- The method introduced by the keyword `rep` which gives the data representation, that is a type called the carrier type. It can be a variable type or a more defined ML like type. This method is mandatory and can be explicitly given or obtained by inheritance.

- Declarations introduced by the keyword `sig` followed by a name and a type.
• Definitions introduced by the keyword let made of a name, a type and an expression. Mutually recursive definitions are introduced by let rec.

• Statements introduced by the keyword property followed by a name and a first-order statement (with the usual connectors e.g. not, and).

• Theorems introduced by the keyword theorem followed by a name, a statement and a proof ultimately checked with the theorem prover Coq.

Statements, definitions and proofs can freely use names of other methods of the species (denoted by self!m). However, in case of dependence cycle, the compiler rejects the species. A let rec definition is not considered as leading to a dependence cycle. Here, we differ a lot from the usual object-oriented languages where methods are considered as mutually recursive without any restriction (indeed a species is not a class, nor an object according to the usual meanings in the object oriented terminology). Method types are type expressions à la ML. The expression self in a type denotes the rep of the species and should be understood as self!rep.

The elements of the species are called entities, to emphasize on the fact that they are defined not only by their representation but also by the functions that manipulate them and their properties.

Inheritance A species B refines a species A if the methods introduced in A and/or the carrier type of A are made more concrete (more defined) in B. In addition, new methods can be added in B and already given definitions can be redefined. This kind of refinement is just a multiple inheritance mechanism. If a species A inherits from several species S_i, our view of data representation implies that all S_i have unifiable rep and that the rep of A is by default the obtained unifier. However, the rep of A can be replaced by a less general unifier (thus more precise). If two S_i have a method having a same name, then their types must be given by the same expression and if they are both defined, we retain the last inherited definition (as in OCaml).

Interfaces The type of a species is obtained by removing definitions and proofs. If self!rep is still a type variable say α, then the species type is prefixed with an existential binder ∃α, which will be eliminated as soon as the rep will be instantiated. The species types remain implicit in the concrete syntax. However, we need here to introduce them to explain the notion of interface. The interface of a species is obtained by abstracting the rep type in all the method types of the species type. This abstraction remains implicit in the concrete syntax as the interface of a species is also denoted by the species name. Interfaces can be ordered by inclusion, a point which gives a very simple notion of subtyping.

Collections A species is said to be complete if all declarations have received definitions and all properties have received proofs. A complete species can be submitted to an abstraction process to create a collection. A collection is indeed
a kind of abstract data type and users have only access to its interface, but having the guarantee that all methods are defined/proven.

**Parametrized species** Species can be parametrized by collections. The formal parameter is introduced by a name \( c \) and an interface \( I \). Any collection \( CC \) having an interface including \( I \) can be used as an actual parameter for \( c \). In the species body, the methods and statements of the parameter are denoted by \( c!m \). The fact that \( CC \) has been created upon a complete species ensures that no link error can arrive at runtime and that proofs of \( CC \) can be used as lemmas.

Species can also be parametrized by entities of collections, themselves introduced as parameters, thus introducing a dependence between parameters. The syntax forbids dependence cycles between parameters.

**Method types** The type of a method is either a basic type like `bool` or a product type or a functional type or a user declared ML-like type. Moreover method types may contains expressions such as `self!rep` and `c!rep` (also written \( c \)). The former denotes an entity of the species, the latter denotes a value of the collection \( c \) and in this case its type is abstract (we do not know anything about the type used to represent it).
3.4 A COMPLETE EXAMPLE

This section illustrates the different concepts of the language and also the programming style with the very classical example of ordered lists. So the aim is to produce a certified tool for manipulating ordered lists that behaves as an abstract data type. In other words, we wish to build a collection of ordered lists. For that, we build a small hierarchy of species and collections drawn in the Figure 3.1 The plan of this section follows this picture bottom up and from left to right. For lack of place, we cannot explain and show all the development, so we give only fragments of species. However, the complete FOCAL code can be found at focal.inria.fr.

**basic_object** Let us first build from scratch a species called basic_object as the root of the hierarchy.

```focal
species basic_object = 
  rep;
  sig parse : string -> self;
  sig print : self -> string;
end
```

**setoid** Species can also be built by inheritance. For instance, let us write the species setoid that describes the notion of setoid, a set equipped with a binary relation equal which is reflexive, symmetric and transitive.

```focal
species setoid inherits basic_object = 
  sig equal in self->self->bool;
  property equal_refl: all x in self, !equal(x,x)
  property equal_sym: all x, y in self, 
     !equal(x,y) -> !equal(y,x);
  property equal_trans: all x, y, z in self, 
     !equal(x,y) -> !equal(y,z) -> !equal(x,z);
  let different(x,y)=#not_b(!equal(x,y));
  theorem different_not_equal: 
    all x, y in self, !different(x,y) <-> (not !equal(x,y))
    proof: def: different;
        {* Intros x y; Unfold abst_different setoid__different;
           Split; Elim (abst_equal x y); Compute; Assumption. *};
end
```

Thus the species setoid inherits the three methods of basic_object and introduces its own methods: the declared function equal and those properties stating that equal is an equivalence relation, the defined function different and a theorem different_not_equal. This last function is defined by using the boolean
negation #not_b on FOCAL booleans\(^1\) and the not yet defined function equal: late binding enters here.

Recall that self in an expression type denotes self!rep and the expression x in self in a formula denotes an entity of the species. The expression !equal is just an abbreviation for self!equal.

The species contains the proof of the theorem different \#not_equal given as a Coq proof script (in italics). We do not go into details on the proof, we just emphasize that the names of the methods are qualified by abst (when the method is only declared) or by species names. This comes from the compilation scheme to Coq that we outline in 3.5.

Before giving the proof script, the programmer has to be precise about the dependencies of the theorem: there is a def-dependence of m upon m' if the former uses the type and the definition of the latter. On the other hand, when a method m only requires the type of another method m', we say that m decl-depends upon m'.

In the present case, different \#not_equal def-depends upon different: the tactic Unfold abst different does unfold the definition of different. A question arises here: what about the proof of the theorem if a daughter species redefines the method different? Because of the def-dependence, the proof will not be valid anymore and it will have to be done again in the new context.

**total_order** Now we define the species partial_order that enriches the setoid notion with an order leq and we continue with the species total_order which constrains the order to be total. Furthermore, this last species defines the strict inequality lt from leq and also the inherited method equal from lt. It is time now to prove the properties of equal declared in setoid.

```coq
species partial_order inherits setoid
sig leq in self -> self -> bool;
property leq_refl: all x, y in self, !equal(x,y) -> !leq(x,y);
property leq_antisymmetric: all x, y in self, !leq(x,y) -> !leq(y,x) -> !equal(x,y) ;
property leq_trans: all x, y, z in self, !leq(x,y) -> !leq(y,z) -> !leq(x,z);
let lt(x,y) = #and_b(!leq(x,y),#not_b(!equal(x,y)));
end

species total_order inherits partial_order
property leq_total: all x, y in self, !leq(x,y) or !leq(y,x);
let equal(x,y) = #and_b(self!leq(x,y),self!leq(y,x));
proof of equal_sym =
def: equal; ...
...
end
```

\(^1\)functions prefixed by \# are predefined functions coming from the FOCAL prelude. Do not confuse #not_b with the unary logical connector written not.
**int\_leq**  To continue the example, we can now define the species \texttt{int\_leq} as a setoid with the usual total order \(\leq\) by inheritance of \texttt{total\_order}. The \texttt{rep} is refined as the \texttt{FOCAL} type \texttt{int}, defined in the \texttt{FOCAL} prelude as a type to be compiled on the \texttt{OCaml} type \texttt{int} and the \texttt{Coq} type \texttt{Z}:

```plaintext
species int\_leq inherits total\_order =
rep = int;
let leq = #int\_leq;
let print = #string\_of\_int;
let parse = #int\_of\_string;
proof of leq\_reflexive = assumed;
proof of leq\_total = assumed;
...
end
```

In this species, \texttt{leq} is defined as the \texttt{OCaml} function \texttt{leq} with the usual order of \texttt{int} (this is the meaning of \texttt{#int\_leq}). At this point, we do not make a correspondence with the \texttt{Coq} \texttt{leq} defined on \texttt{Z}. Thus, we have to prove that the \texttt{OCaml} function \texttt{leq} is indeed a total order. Here we have two possibilities. Either we try to do the proofs and we have to embark on studying machine integer implementation, a point which is very far from our initial problem. Or we consider that integer machine implementation is trusted and we do the proofs by just answering \texttt{assumed}. That means that the statement will be considered by \texttt{Coq} as an axiom. Doing a proof by \texttt{assumed} is a pragmatical choice based on the application domain and in the confidence we can have on some properties (according to the context, do we need to formally prove some Pythagorean theorem?) and also on the balance between trust and costs. Here testing the property by evaluating it with some pertinent test cases could strengthen our confidence in the property.

**int\_leq\_co**  Returning to the species \texttt{int\_leq}, we can consider it as complete: methods are all defined/proved. Thus we are allowed (by the compiler) to build from it a \texttt{collection} which encapsulates the \texttt{rep} type and provides the functions defined in the species \texttt{int\_leq}. A collection is indeed a certified abstract data type, its interface being the one of the complete species from which it has been built.

```plaintext
collection int\_leq\_co implements int\_leq;
```

**fin\_set**  At this stage, we want to introduce the notion of finite set. First we introduce the species \texttt{fin\_set} parametrized by a collection \texttt{a} satisfying the interface \texttt{setoid}: \texttt{a} must provide all the operations \texttt{parse, print, equal} which handle values of an abstract data type denoted in the body of the species by \texttt{a}. The species declares the set primitives operations. Each of them is completed by its specification (e.g. \texttt{singleton} and \texttt{singleton\_spec}). The species inherits from \texttt{partial\_order: leq} is defined as the subset predicate and is proved a partial order.
species fin_set (a is setoid) inherits partial_order =
sig empty in self;
sig singleton in a -> self;
sig isin in a -> self -> bool;
sig cardinal in self -> int;
sig subset in self -> self -> bool;
...
let leq = !subset;

property singleton_spec: all x, y in a,
  !est_element(y, !singleton(x)) <-> a!equal(x,y);
...

**ordered_set**  We now build the species *ordered_set* that specifies the finite
sets the elements of which are taken in a totally ordered set: it declares a new
method `min` that returns the minimal element of a non empty set.

species ordered_set (a is total_order) inherits fin_set (a) =
sig min in self -> a;
property min_spec : all e in self,
  not(!empty(e))->(!isin(!min(e),e) and
  (all x in a, !isin(x,e) -> a!leq(!min(e),x)));
end

**llist**  We write the complete species *llist* of finite lists. A finite list is
considered as a finite set, so this new species inherits from *fin_set(a)*. We
choose to represent our lists as lists à la ML constrained to contain distinct el-
ements. The `rep` type is then `list(a)` that is the type constructor `list` (the
same as in OCaml) applied to the rep of the collection `a`, named also `a`. Some
of the primitives declared in `llist(a)`, e.g. `empty`, `singleton` are defined as
operations upon lists while `union`, `diff` and `inter` are kept as signatures. The
defined primitives are proved correct with respect to their specifications. Further-
more, they are proved correct with respect to the *representation invariant*, that is
the property `well-formed` establishing that the representation of a set is a `well-
formed` list containing distinct elements. For instance we prove that `!empty` is
well-formed and `subset` preserves the well-formedness etc. Up to now, these
properties are written manually, we plan to generate them automatically from the
representation invariant.

species llist (a is setoid) inherits fin_set (a)=
rep = list (a);
(* representation invariant *)
let rec wf (l in self) =
  match l with
    [] -> #True
  | a::l -> #and_b (#not_b (!isin (a, l)), !wf (l))
end;
let empty in self = []; let singleton(x) = [x]; let isin(x,e) = match e with
    []  -> #False
  | y::l -> if a!equal(x,y) then #True else !isin(x,l)
end;

proof of singleton_spec = ...

theorem wellformed_empty: !wf(!empty)
  proof: ...
end

If we derive from llist a species implementing and proving correct the operations inter, union and diff, this derived species once instantiated would be a candidate to build a collection, e.g. the collection of lists of integers.

ordered_list To achieve our initial objective, it remains to write the species ordered_list parametrized by a collection of interface total_order: it implements the ordered lists and the associated operations, in particular the union function as the merge of two ordered lists. This species uses multiple inheritance: ordered_list(a) inherits ordered_set(a) and llist(a). So the rep is inherited from llist(a) and is list(a). The min function inherited from ordered_set(a) is defined as the function returning the head of the list.

The representation invariant is updated: we add the fact that the elements are ordered according to the total order of a. So we have to prove that each operation is correct with respect to this new property. The first invariant (wf) is still appropriate, so we have to state that inter, union and diff preserve this property. The proof part of this species is important but also heavy.

species ordered_list (a is total_order)
  inherits ordered_set(a), llist(a) =
  (* representation invariant *)
let rec ordered(l in self) in bool =
  match l with
    []  -> #True
  | [x] -> #True
  | x::y::l -> #and_b(a!lt(x,y), !ordered(l))
end;

theorem ordered_empty : !ordered(!empty)
  ...
theorem wellformed_union : all s, t in self,
  !wf(s) -> !wf(t) -> !wf(!union(s,t))
  ...
theorem ordered_union : all s, t in self,
  !ordered(s) -> !ordered(t) -> !ordered(!union(s,t))
  ...
end
Chapter 3. Building Certified Components within FOCAL

ordered_list_int  To end with the hierarchy built in this section, it remains for example to instantiate this species with the previous collection int_leq and then to produce the corresponding collection.

```plaintext
species ordered_list_int inherits ordered_list(int_leq) =
end
collection ordered_list_int_co implements ordered_list_int;
```

3.5 COMPILING

The FOCAL development environment comes with a compiler which starts with a sequence of static analyses and then produces different outputs [Pre03]. The Figure 3.2 illustrates the different tools that the environment includes.

Analysis  After a syntactic analysis, the compiler first performs a typing analysis of methods, with a very simple Damas-Milner like algorithm. Then, it constructs a dependence graph between all methods, inherited or native to try to order consistently the methods. In case of a dependence cycle, the compiler fails. It checks if some methods are redefined and removes all proofs having def-dependencies on these methods. It also checks that the species used to build a collection is indeed complete. And, it verifies that actual parameters have an interface fitting with the interface of the formal parameter. When the species is recognized as correct, it enters the translation phase, mostly based on the dependence graph. The different possible translations are independent.

Running  The compiler produces a source code for OCaml using classes, objects and modules. The obtained object code is very efficient [BHR01] (time and memory) thanks to the OCaml implementation.

Proving  The compiler produces Coq code corresponding to the FOCAL components, that is processed by Coq in order to check the proofs. The Coq translation
scheme relies on the following features:

- A species is translated into a record whose fields are all the methods of the species, inherited, declared and defined.

- A method \(m\) defined in a species \(s\) is translated into a method generator \(\text{gen}_m\) that is a Coq function/property parametrized by the methods upon which \(m\) depends.

- And in the species which inherits \(m\), the corresponding field is built by application of \(\text{gen}_m\).

So in the Coq translation, inheritance is flatten and dependencies are made explicit. For further explanations the reader can consult [PD 02, PDH02, Pre03]. Typechecking with Coq ensures that the proofs are valid but also that design stages are coherent and that all methods are well-typed.

There are two ways to do the proofs. Either, they are done with the Coq tactics as is exemplified in 3.4 Or the programmer may discharge them with the help of Zenon, a very recent dedicated prover [Do104]. Zenon is a first order theorem prover, developed by Doligez, based on the tableau method incorporating implementation novelties such as sharing. Inspired by the Lamport’s hierarchical style it is declarative so it means that the user gives intermediate lemmas as hints and Zenon combines them to search the proof. Furthermore Zenon can do proof reconstruction: once the proof is found, it can be output in a Coq-checkable format to be typechecked by Coq.

As already mentioned in Section 3.4 the redefinition of a function by inheritance may invalidate the proofs that involve this function. The proofs are then erased and must be done again in the new context. Thus the main difficulty is to choose the best node in the hierarchy to do a proof. The naive view which consists in proving the properties as soon as possible is not always the right methodology: the proofs are done at an abstract level and the method may not be used in the implementations. The alternative solution that is to prove everything when completing a species is definitely not the right solution either: there are too much implementation details that complicate the proofs and the user may be led to duplicate the proofs. In [PJ03], Prevosto and Jaume propose a coding style to minimize the number of proofs to be redone in the case of a redefinition: e.g. minimize number of theorems with def-dependencies, minimize complexity of theorems with def-dependencies and do not have more than one def-dependence per theorem. The tools of the FOCAL environment to visualize the dependencies may help in such a context.

**Documenting** The tool called FOCDOC [MP03] automatically generates XML documentation, thus the documentation of a component is always coherent with respect to its implementation. This tool uses its own XML format that contains information coming not only from structured comments and FOCAL concrete syntax but also from type inference and dependence analysis. From this XML representation, FOCDOC generates HTML files or Latex files.
Chapter 3. Building Certified Components within FOCAL

3.6 APPLICATIONS

**Computer algebra**  FOCAL has been first used to develop a library for computer algebra, a subject which indeed does not exemplify critical systems. But it has the advantage that its formal specification can be found in mathematical books and that its algorithms may be very intricate, use complex data structures and are heavy time and memory consuming. So, this is a very good running example. The library offers powerful tools to manipulate multivariate polynomials. The specification stage introduces all needed mathematical notions, the design stage introduces some data representations and algorithms and the implementation offers different implementations of polynomials according to the chosen representation of degrees and coefficients (machine integers, arbitrary precision integers or rationals, for example). Classical benchmarks on polynomials (such as to compute a sub-resultant -a kind of gcd- of \( P = x^{30} + ax^{20} + 2ax^{10} + 3a \) and \( Q = x^{25} + 4bx^{15} + 5bx^{5} \) where \( a \) is a natural between \( 10^{100} \) and \( 10^{700} \) and \( b = a + 1 \)) have shown very good efficiency of the runnable code [BHR01].

The automatic documentation viewer gives a precise description of the underlying mathematical structures and the implementation choices. Furthermore FOCAL also proposes dedicated tools to translate OMDoc documents (Open Maths Document, a standard format for encoding mathematical content) into FOCAL species or collections and vice versa.

**Security policies**  Currently, we use FOCAL to build a library for expressing and coding security policies [JM05]. The specification stage introduces subjects, objects and access rights. Then frameworks are built upon finite subsets of subjects, objects and access rights, adding a lattice of security levels. Models are built upon frameworks by adding some evolution rules of access rights. A security system supervising the application of security rules is just an implementation of a model, by a kind of state machine, essentially defined by its transition function. We have used this library to implement a supervisor for SQL requests control in a relational database, using the well-known Bell-Lapadula system as a security policy.

**Airport Security**  The security of civil aviation is governed by international standards and recommended practices that detail the responsibilities of the different actors such as states, operators and security agents. These security rules are described in natural language documents from a few dozen to several hundred pages. These documents give the specifications of various procedures and artifacts which implement security in airports, aircrafts and air traffic control. It is essential to ensure the consistency and completeness of these specifications. The project Edemoi [Ede05] aims at constructing and analyzing a precise reference document that models and structures the current international standards and recommended practices about airport security. A first step consisted in reading the documents, especially the Annex 17 [Ann02] and modelling it with a graphical UML based formalism. This first model is a basis for discussion and validation with the certification authority, namely here the International Civil Aviation Organisation (ICAO). The second step of the Edemoi approach is to provide formal
models and use specialized tools to study their consistency. The chosen languages for this stage are Z, B and FOCAL. The FOCAL modelling of the annex 17 is in progress and emphasizes again the great flexibility brought by the possibility to intermingle specifications and code. Indeed in this model some properties are very abstract while some others are security procedures.

3.7 RELATED WORK

FOCAL proposes a rather original way of using object-oriented features, modularity and dependent types. There are a lot of proposals and languages which combine more or less these three features, for instance modules, mixins, classes, EML (extended ML). Except in Coq [Chr03], modules do not mix code, specifications and proofs. Several languages propose functors to use modules as parameters of other modules. They can mimic some kind of inheritance, but it is quite poor and we lose readability. Mixins add some object-oriented features but, as far as we know, do not accept true redefinition of functions. EML [KST97] is a language developed some years ago to specify and develop modular SML programs. Thus Focal can be considered very close to EML but goes further by providing the proof framework. As far as we know, this aspect is not achieved in EML although some work exists about the termination of functions (a proof obligation generator for PVS).

With FOCAL, the verification of the properties can be done at different steps in the development process, even delayed until the end of the development making FOCAL a code verifier. A complete species is a kind of object-oriented program where the methods are annotated with properties, resembling Java programs annotated with JML. Such species also contain proofs. Some tools, e.g. Krakatoa [MPMU04], provide static verification of JML assertions, with different levels of expressiveness and automation. Krakatoa allows to prove that a JML-annotated method of a Java program meets its specifications given as a JML interface. This amounts to proving that class invariants, as well as post-conditions, hold at the end of a method execution, provided that invariants and pre-conditions were valid at the beginning. But these tools are external tools.

The Atelier B [AtB05] is a complete development environment allowing one to go from specification to code by proved correct refinement steps. This environment and more generally the B method [Abr96], and FOCAL are tools to realise formal conception. Furthermore, both enable to combine components. But the FOCAL object-oriented features together with the abstraction mechanisms (parameters and collections) give it an expressiveness and a flexibility that the importation rules of B do not provide.

3.8 CONCLUSION AND FUTURE WORK

FOCAL is a framework and a set of tools for developing executable and certified code. It is also a language to be used from the specification stage to the implementation one. Confidence is achieved by different means: the OCaml generated code
is verified by typing tools; dependencies are also verified by Coq; each declared property in a collection is proved or assumed (hypothesis explicitly appearing in the documentation); proof terms are produced and ultimately verified by Coq; documentation is automatically produced, and thus always complete and coherent with respect to the development. Moreover the specification of the language itself has been done in Coq and the correctness of the compiler against this specification is proved (by hand) in [PB05].

Future work focuses on two principal directions: proof search and specification style. Proving is very often fastidious and painful. So it is necessary, even fundamental, to provide some kind of automation. Our first axis consists in integrating rewriting tools and deduction modulo features into FOCAL. This work is done in the context of the project Modulogic [Mod05].

The second concern, specification style, is undertaken from different points: experimenting with FOCAL in other application domains, such as in the Edemoi project, adding the notion of state and taking into account behavioural properties. Indeed the core language of FOCAL is purely functional: the species have no state; no imperative feature is allowed. This language design choice was entailed by the need of confidence and to ease proofs. However in some applications for a matter of efficiency or readability, the notions of time and state are mandatory: methods become operations that may access or update the state according to an invariant property for example. A possible approach could be to incorporate some state-oriented constructs in the FOCAL language while still relying on a general functional framework (by computing effects for example).

FOCAL does not take into account neither time nor any temporal property. This is the subject of the complementary project Alidecs that started very recently with a synchronous approach [Ali05].

Acknowledgements. The authors wish to thank all the FOCAL team (from labs. CEDRIC, LIP6 and INRIA) for helpful discussions in particular about the central example of this paper. We are grateful to the anonymous reviewers for their constructive remarks.

REFERENCES


Chapter 4

Calculating an Exceptional Machine

Graham Hutton, Joel Wright

Abstract: In previous work we showed how to verify a compiler for a small language with exceptions. In this article we show how to calculate, as opposed to verify, an abstract machine for this language. The key step is the use of Reynold’s defunctionalization, an old program transformation technique that has recently been rejuvenated by the work of Danvy et al.

4.1 INTRODUCTION

Exceptions are an important feature of modern programming languages, but their compilation has traditionally been viewed as an advanced topic. In previous work we showed how the basic method of compiling exceptions using stack unwinding can be explained and verified using elementary functional programming techniques [HW04]. In particular, we developed a compiler for a small language with exceptions, together with a proof of its correctness.

In the formal reasoning community, however, one prefers constructions to verifications [Bac03]. That is, rather than first writing the compiler and then separately proving its correctness with respect to a semantics for the language, it would be preferable to try and calculate the compiler [Mei92] directly from the semantics, with the aim of giving a systematic discovery of the idea of compiling exceptions using stack unwinding, as opposed to a post-hoc verification.

In this article we take a step towards this goal, by showing how to calculate an abstract machine for evaluating expressions in our language with exceptions. The key step in the calculation is the use of defunctionalization, a program transformation technique that eliminates the use of higher-order functions, first introduced by Reynolds in his seminal work on definitional interpreters [Rey72].
Despite being simple and powerful, defunctionalization seems to be somewhat neglected in recent years. For example, it features in few modern courses, textbooks, and research articles on program transformation, and does not seem to be as widely known and used as it should be. Recently, however, defunctionalization has been rejuvenated by the work of Danvy et al., who show how it can be applied in a variety of different areas, including the systematic design of abstract machines for functional languages [DN01, ABDM03b, ADM04].

In this article, we show how Danvy’s approach can be used to calculate an abstract machine for our language with exceptions. Moreover, the calculation is rabbit free, in the sense that there are no Eureka steps in which one needs to metaphorically pull a rabbit out of a hat — all the required concepts arise naturally from the calculation process itself. The approach is based upon the work of Danvy et al., but the emphasis on calculation and the style of exposition are our own.

The language that we use comprises just integer values, an addition operator, a single exceptional value called throw, and a catch operator for this value [HW04]. This language does not provide features that are necessary for actual programming, but it does provide just what we need for expository purposes. In particular, integers and addition constitute a minimal language in which to consider computation using a stack, and throw and catch constitute a minimal extension in which such computations can involve exceptions.

Our development proceeds in two steps, starting with the exception-free part of the language to introduce the basic techniques, to which support for exceptions is then added in the second step. All the programs are written in Haskell [Pey03], and all the calculations are presented using equational reasoning. An extended version of the article that includes the calculations omitted here for reasons of space is available from www.cs.nott.ac.uk/~gmh/machine-extended.pdf.

### 4.2 Abstract Machines

An abstract machine can be defined as a term rewriting system for executing programs in a particular language, and is given by a set of rewrite rules that make explicit how each step of execution proceeds. Perhaps the best known example is Landin’s SECD machine for the lambda calculus [Lan64], which comprises a set of rewrite rules that operate on tuples with four components that give the machine its name, called the stack, environment, control and dump.

For a simpler example, consider a language in which programs comprise a sequence of push and add operations on a stack of integers. In Haskell, such programs, operations and stacks can be represented by the following types:

\[
\begin{align*}
\text{type } \text{Prog} &= [\text{Op}] \\
\text{data } \text{Op} &= \text{PUSH Int} \mid \text{ADD} \\
\text{type } \text{Stack} &= [\text{Int}]
\end{align*}
\]

An abstract machine for this language is given by defining two rewrite rules on pairs of programs and stacks from the set $\text{Prog} \times \text{Stack}$:
Chapter 4. Calculating an Exceptional Machine

\[
\begin{align*}
\langle \text{PUSH } n : \text{ops} , s \rangle & \quad \rightarrow \quad \langle \text{ops} , n:s \rangle \\
\langle \text{ADD} : \text{ops} , n:m:s \rangle & \quad \rightarrow \quad \langle \text{ops} , n+m : s \rangle 
\end{align*}
\]

The first rule states that push places a new integer on top of the stack, while the second states that add replaces the top two integers on the stack by their sum. This machine can be implemented in Haskell by an execution function that repeatedly applies the two rules until this is no longer possible:

\[
\begin{align*}
\text{exec} & :: (\text{Prog}, \text{Stack}) \rightarrow (\text{Prog}, \text{Stack}) \\
\text{exec} (\text{PUSH } n : \text{ops}, s) & = \text{exec (ops}, n:s) \\
\text{exec} (\text{ADD} : \text{ops}, n:m:s) & = \text{exec (ops}, n+m : s) \\
\text{exec} (p, s) & = (p, s)
\end{align*}
\]

For example, \(\text{exec ([PUSH 1, PUSH 2, ADD], [])}\) gives the result \(([[], [3]])\). In the remainder of this article, we will use the term abstract machine for such a functional implementation of an underlying set of rewrite rules.

### 4.3 ARITHMETIC EXPRESSIONS

As in our previous work \[HW04\], let us begin our development by considering a simple language of expressions comprising integers and addition, whose semantics is given by a function that evaluates an expression to its integer value:

\[
\begin{align*}
\text{data Expr} & = \text{Val Int} | \text{Add Expr Expr} \\
\text{eval} & :: \text{Expr} \rightarrow \text{Int} \\
\text{eval (Val } n \text{)} & = n \\
\text{eval (Add } x \text{ } y \text{)} & = \text{eval } x + \text{eval } y
\end{align*}
\]

We will now calculate an abstract machine for this language, by making a series of three transformations to the semantics.

**Step 1: Add continuations**

At present, the order in which addition evaluates its argument expressions is determined by the language in which the semantics is written, in this case Haskell. The first step in producing an abstract machine is to make the order of evaluation explicit in the semantics itself. A standard technique for achieving this aim is to rewrite the semantics in **continuation-passing style** \[Rey72\].

A **continuation** is a function that will be applied to the result of an evaluation. For example, in the equation \(\text{eval (Add } x \text{ } y \text{)} = \text{eval } x + \text{eval } y\) from our semantics, when the first recursive call, \(\text{eval } x\), is being evaluated, the remainder of the right-hand side, \(+ \text{eval } y\), can be viewed as a continuation for this evaluation, in the sense that it is the function that will be applied to the result.

More formally, in the context of our semantics \(\text{eval} :: \text{Expr} \rightarrow \text{Int}\), a continuation is a function of type \(\text{Int} \rightarrow \text{Int}\) that will be applied to the result of type \(\text{Int}\) to give a new result of type \(\text{Int}\). (This type can be generalised to \(\text{Int} \rightarrow \text{a}\), but we
do not need the extra generality for our purposes here.) We capture the notion of such a continuation using the following type definition:

\[
\text{type } \text{Cont} = \text{Int} \rightarrow \text{Int}
\]

Our aim now is to define a new semantics, \(\text{eval}'\), that takes an expression and returns an integer as previously, but also takes a continuation that will be applied to the resulting integer. That is, we seek to define a function

\[
\text{eval}' :: \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Int}
\]

such that:

\[
\text{eval}' e \ c = c (\text{eval} e)
\]

At this point in most texts, a recursive definition for \(\text{eval}'\) would normally be written and then either proved to satisfy the above equation, or this be justified by appealing to the correctness of a general continuation-passing transformation. However, we prefer to calculate the definition for \(\text{eval}'\) directly from the above equation, by the use of structural induction on \(\text{Expr}\).

Case: \(\text{Val} \ n\)

\[
\text{eval}' (\text{Val} \ n) \ c = c \ n
\]

Case: \(\text{Add} \ x \ y\)

\[
\text{eval}' (\text{Add} \ x \ y) \ c = c (\text{eval} (\text{Add} \ x \ y)) = \text{eval}' x (\lambda n \rightarrow \text{eval}' y (\lambda m \rightarrow c (n + m)))
\]

In conclusion, we have calculated the following recursive definition:

\[
\text{eval}' :: \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Int}
\]

\[
\text{eval}' (\text{Val} \ n) \ c = c \ n
\]

\[
\text{eval}' (\text{Add} \ x \ y) \ c = \text{eval}' x (\lambda n \rightarrow \text{eval}' y (\lambda m \rightarrow c (n + m)))
\]
That is, for an integer value we simply apply the continuation, while for an addition we evaluate the first argument and call the result $n$, then evaluate the second argument and call the result $m$, and finally apply the continuation to the sum of $n$ and $m$. In this manner, order of evaluation is now explicit in the semantics.

Note that we have ensured that addition evaluates its arguments from left-to-right by first abstracting over $eval\ x$ in the above calculation, and then abstracting over $eval\ y$. It would be perfectly valid to proceed in the other direction, which would result in right-to-left evaluation. Note also that our original semantics can be recovered from our new semantics, by substituting the identity continuation $\lambda n \rightarrow n$ into the equation from which $eval'$ was constructed. That is, our original semantics $eval$ can now be redefined as follows:

\[
\begin{align*}
  eval & \quad :: \quad Expr \rightarrow Int \\
  eval\ e & = \quad eval'\ e\ (\lambda n \rightarrow n)
\end{align*}
\]

**Step 2: Defunctionalize**

We have now taken a step towards an abstract machine by making evaluation order explicit, but in so doing have also taken a step away from such a machine by making the semantics into a higher-order function. The next step is to regain the first-order nature of the original semantics by eliminating the use of continuations, but retaining the explicit order of evaluation that they introduced.

A standard technique for eliminating the use of functions as arguments is *de-functionalization* [Rey72]. This technique is based upon the observation that we do not usually need the entire function-space of possible argument functions, because only a few forms of such functions are actually used in practice. Hence, we can represent the argument functions that we actually need using a datatype, rather than using the actual functions themselves.

In our new semantics, there are only three forms of continuations that are actually used, namely one to invoke the semantics, and two in the case for evaluating an addition. We begin by separating out these three forms, by giving them names and abstracting over their free variables. That is, we define three combinators for constructing the required forms of continuations:

\[
\begin{align*}
  c1 & \quad :: \quad Cont \\
  c1 & = \quad \lambda n \rightarrow n \\
  c2 & \quad :: \quad Expr \rightarrow Cont \rightarrow Cont \\
  c2\ y\ c & = \quad \lambda n \rightarrow eval'\ y\ (c3\ n\ c) \\
  c3 & \quad :: \quad Int \rightarrow Cont \rightarrow Cont \\
  c3\ n\ c & = \quad \lambda m \rightarrow c\ (n+m)
\end{align*}
\]

At present we have just used anonymous names $c1$, $c2$ and $c3$ for the combinators, but these will be replaced by more suggestive names later on. Using these
combinators, our semantics can now be rewritten as follows:

\[
\begin{align*}
\text{eval}' & :: \text{Expr} \to \text{Cont} \to \text{Int} \\
\text{eval}' (\text{Val} \ n) \ c &= c \ n \\
\text{eval}' (\text{Add} \ x \ y) \ c &= \text{eval}' \ x \ (c2 \ y \ c) \\
\text{eval} & :: \text{Expr} \to \text{Int} \\
\text{eval} \ e &= \text{eval}' \ e \ c1
\end{align*}
\]

The next stage in applying defunctionalization is to define a datatype whose values represent the three combinators that we have isolated:

\[
\text{data} \ \text{CONT} = \text{C1} | \text{C2} \ \text{Expr} \ \text{CONT} | \text{C3} \ \text{Int} \ \text{CONT}
\]

The constructors of this datatype have the same types as the corresponding combinators, except that the new type \text{CONT} plays the role of \text{Cont}:

\[
\begin{align*}
\text{C1} & :: \text{CONT} \\
\text{C2} & :: \text{Expr} \to \text{CONT} \to \text{CONT} \\
\text{C3} & :: \text{Int} \to \text{CONT} \to \text{CONT}
\end{align*}
\]

The fact that values of type \text{CONT} represent continuations of type \text{Cont} can be formalised by defining a function that maps from one to the other:

\[
\begin{align*}
\text{apply} & :: \text{CONT} \to \text{Cont} \\
\text{apply} \ \text{C1} &= c1 \\
\text{apply} \ (\text{C2} \ y \ c) &= c2 \ y \ (\text{apply} \ c) \\
\text{apply} \ (\text{C3} \ n \ c) &= c3 \ n \ (\text{apply} \ c)
\end{align*}
\]

The name of this function derives from the fact that when its type is expanded to \text{apply} :: \text{CONT} \to \text{Int} \to \text{Int}, it can be viewed as applying a representation of a continuation to an integer to give another integer.

Our aim now is to define a new semantics, \text{eval''}, that behaves in the same way as our previous semantics, \text{eval'}, except that it uses values of type \text{CONT} rather than continuations of type \text{Cont}. That is, we seek to define a function

\[
\text{eval''} :: \text{Expr} \to \text{CONT} \to \text{Int}
\]

such that:

\[
\text{eval''} \ e \ c = \text{eval'} \ e \ (\text{apply} \ c)
\]

As previously, we calculate the definition for the function \text{eval''} directly from this equation by the use of structural induction on \text{Expr}.

Case: \text{Val} \ n

\[
\begin{align*}
\text{eval''} (\text{Val} \ n) \ c &= \{ \ \text{specification of eval''} \ \} \\
\text{eval'} (\text{Val} \ n) \ (\text{apply} \ c) &= \{ \ \text{definition of eval'} \ \} \\
\text{apply} \ c \ n
\end{align*}
\]
Case: $Add \ x \ y$

\[
\text{eval}''(Add \ x \ y) \ c = \{ \text{specification of eval}'' \}
\]

\[
\text{eval}'(Add \ x \ y) \ (apply \ c) = \{ \text{definition of eval}' \}
\]

\[
\text{eval}' \ x \ (c2 \ y \ (apply \ c)) = \{ \text{definition of apply} \}
\]

\[
\text{eval}' \ x \ (apply \ (C2 \ y \ c)) = \{ \text{induction hypothesis for x} \}
\]

\[
\text{eval}'' \ x \ (C2 \ y \ c)
\]

In conclusion, we have calculated the following recursive definition:

\[
\text{eval}'' :: \ Expr \rightarrow \ CONT \rightarrow \ Int
\]

\[
\text{eval}'' (\text{Val} \ n) \ c = \text{apply} \ c \ n
\]

\[
\text{eval}'' (Add \ x \ y) \ c = \text{eval}'' \ x \ (C2 \ y \ c)
\]

However, the definition for apply still refers to the previous semantics eval', via its use of the combinator c2. We calculate a new definition for apply that refers to our new semantics instead by the use of case analysis on CONT.

Case: $C1$

\[
\text{apply} \ C1 \ n = \{ \text{definition of apply} \}
\]

\[
c1 \ n = \{ \text{definition of c1} \}
\]

\[
n
\]

Case: $C2 \ y \ c$

\[
\text{apply} \ (C2 \ y \ c) \ n = \{ \text{definition of apply} \}
\]

\[
c2 \ y \ (apply \ c) \ n = \{ \text{definition of c2} \}
\]

\[
\text{eval}' \ y \ (c3 \ n \ (apply \ c)) = \{ \text{definition of apply} \}
\]

\[
\text{eval}' \ y \ (apply \ (C3 \ n \ c)) = \{ \text{specification of eval}'' \}
\]

\[
\text{eval}'' \ y \ (C3 \ n \ c)
\]

Case: $C3 \ n \ c$

\[
\text{apply} \ (C3 \ n \ c) \ m = \{ \text{definition of apply} \}
\]

\[
c3 \ n \ (apply \ c) \ m = \{ \text{definition of c3} \}
\]

\[
\text{apply} \ c \ (n + m)
\]
In conclusion, we have calculated the following new definition:

\[
\begin{align*}
\text{apply} & :: \text{CONT} \to \text{Int} \to \text{Int} \\
\text{apply } C1 \ n & = n \\
\text{apply } (C2 \ y \ c) \ n & = \text{eval}'' \ y \ (C3 \ n \ c) \\
\text{apply } (C3 \ n \ c) \ m & = \text{apply} \ c \ (n + m)
\end{align*}
\]

We have now eliminated the use of functions as arguments, and hence made the semantics first-order again. But what about the fact that \text{eval}'' and \text{apply} are curried functions, and hence return functions as results? As is common practice, we do not view the use of functions as results as being higher-order, as it is not essential and can easily be eliminated if required by uncurrying.

Finally, our original semantics can be recovered from our new semantics by redefining \text{eval} e = \text{eval}'' \ e \ C1, as can be verified by a simple calculation:

\[
\begin{align*}
\text{eval} \ e & = \{ \text{previous definition of eval} \} \\
\text{eval}'' \ e \ (\lambda n \to n) & = \{ \text{definition of } C1 \} \\
\text{eval}'' \ e \ C1 & = \{ \text{definition of } \text{apply} \} \\
\text{eval}'' \ e \ (\text{apply} \ C1) & = \{ \text{specification of eval}'' \}
\end{align*}
\]

**Step 3: Refactor**

At this point, after making two transformations to the original semantics, the reader may be wondering what we have actually produced? In fact, we now have an abstract machine for evaluating expressions, but this only becomes clear after we refactor the definitions, in this simple case by just renaming the components. In detail, we rename \text{CONT} as \text{Cont}, \text{C1} as \text{STOP}, \text{C2} as \text{EVAL}, \text{C3} as \text{ADD}, \text{eval}'' as \text{eval}, \text{apply} as \text{exec}, and \text{eval} as \text{run} to give the following machine:

\[
\begin{align*}
data \text{ Cont} & = \text{STOP} \mid \text{EVAL} \text{ Expr} \text{ Cont} \mid \text{ADD} \text{ Int} \text{ Cont} \\
\text{eval} & :: \text{Expr} \to \text{Cont} \to \text{Int} \\
\text{eval} \ (\text{Val} \ n \ c) & = \text{exec} \ c \ n \\
\text{eval} \ (\text{Add} \ x \ y \ c) & = \text{eval} \ x \ (\text{EVAL} \ y \ c) \\
\text{exec} & :: \text{Cont} \to \text{Int} \to \text{Int} \\
\text{exec} \text{ STOP} \ n & = n \\
\text{exec} \ (\text{EVAL} \ y \ c) \ n & = \text{eval} \ y \ (\text{ADD} \ n \ c) \\
\text{exec} \ (\text{ADD} \ n \ c) \ m & = \text{exec} \ c \ (n + m) \\
\text{run} & :: \text{Expr} \to \text{Int} \\
\text{run} \ e & = \text{eval} \ e \ \text{STOP}
\end{align*}
\]

We now explain the four parts of the abstract machine in turn:
**Cont** is the type of control stacks for the machine, containing instructions that determine the behaviour of the machine after evaluating the current expression. The meaning of the three forms of instructions, *STOP*, *EVAL* and *ADD* will be explained shortly. Note that the type of control stacks could itself be refactored as an explicit list of instructions, as follows:

\[
\begin{align*}
\text{type } & \text{Cont } = [\text{Inst}] \\
\text{data } & \text{Inst } = \text{ADD Int} \mid \text{EVAL Expr}
\end{align*}
\]

However, we prefer the original definition above because it only requires the definition of a single type rather than a pair of types.

- **eval** evaluates an expression in the context of a control stack. If the expression is an integer value, it is already fully evaluated, and we simply execute the control stack using this integer as an argument. If the expression is an addition, we evaluate the first argument, \(x\), placing the instruction *EVAL* \(y\) on top of the current control stack to indicate that the second argument, \(y\), should be evaluated once that of the first argument is completed.

- **exec** executes a control stack in the context of an integer argument. If the stack is empty, represented by the instruction *STOP*, we simply return the integer argument as the result of the execution. If the top of the stack is an instruction *EVAL* \(y\), we evaluate the expression \(y\), placing the instruction *ADD* \(n\) on top of the remaining stack to indicate that the current integer argument, \(n\), should be added together with the result of evaluating \(y\) once this is completed. Finally, if the top of the stack is an instruction *ADD* \(m\), evaluation of the two arguments of an addition is now complete, and we execute the remaining control stack in the context of the sum of the two resulting integers.

- **run** evaluates an expression to give an integer, by invoking **eval** with the given expression and the empty control stack as arguments.

The fact that our machine uses two mutually recursive functions, **eval** and **exec**, reflects the fact that it has two states, depending upon whether it is being driven by the structure of the expression (**eval**) or the control stack (**exec**). To illustrate the machine, here is how it evaluates \((2 + 3) + 4\):

\[
\begin{align*}
\text{run } (\text{Add} (\text{Add} (\text{Val} 2) (\text{Val} 3)) (\text{Val} 4))
&= \text{eval } (\text{Add} (\text{Add} (\text{Val} 2) (\text{Val} 3)) (\text{Val} 4)) \text{ STOP} \\
&= \text{eval } (\text{Add} (\text{Val} 2) (\text{Val} 3)) (\text{EVAL} (\text{Val} 4) \text{ STOP}) \\
&= \text{eval } (\text{Val} 2) (\text{EVAL} (\text{Val} 3) (\text{EVAL} (\text{Val} 4) \text{ STOP})) \\
&= \text{exec } (\text{EVAL} (\text{Val} 3) (\text{EVAL} (\text{Val} 4) \text{ STOP})) 2 \\
&= \text{eval } (\text{Val} 3) (\text{ADD} 2 (\text{EVAL} (\text{Val} 4) \text{ STOP})) \\
&= \text{exec } (\text{ADD} 2 (\text{EVAL} (\text{Val} 4) \text{ STOP})) 3 \\
&= \text{exec } (\text{EVAL} (\text{Val} 4) \text{ STOP}) 5 \\
&= \text{eval } (\text{Val} 4) (\text{ADD} 5 \text{ STOP}) \\
&= \text{exec } (\text{ADD} 5 \text{ STOP}) 4 \\
&= \text{exec } \text{STOP} 9 \\
&= 9
\end{align*}
\]
Note how the function $eval$ proceeds downwards to the leftmost integer in the expression, maintaining a trail of the pending right-hand expressions on the control stack. In turn, the function $exec$ then proceeds upwards through the trail, transferring control back to $eval$ and performing additions as appropriate.

Readers familiar with Huet's zipper data structure for navigating around expressions [Hue97] may find it useful to note that our type $Cont$ is a zipper data structure for $Expr$, specialised to the purpose of evaluating expressions. Moreover, this specialised zipper arose naturally here by a process of systematic calculation, and did not require any prior knowledge of this structure.

### 4.4 ADDING EXCEPTIONS

Now let us extend our language of arithmetic expressions with simple primitives for throwing and catching an exception:

$$
\text{data } Expr = \ldots | \text{Throw} | \text{Catch } Expr \text{ Expr}
$$

Informally, $Throw$ abandons the current computation and throws an exception, while $Catch x y$ behaves as the expression $x$ unless it throws an exception, in which case the catch behaves as the handler expression $y$. To formalise the meaning of these new primitives, we first recall the $Maybe$ type:

$$
\text{data } Maybe a = \text{Nothing} | \text{Just } a
$$

That is, a value of type $Maybe a$ is either $Nothing$, which we think of as an exceptional value, or has the form $Just x$ for some $x$ of type $a$, which we think of as a normal value [Spi90]. Using this type, our original semantics for expressions can be rewritten to take account of exceptions as follows:

$$
\text{eval} :: Expr \rightarrow Maybe Int
\text{eval } (\text{Val } n) = \text{Just } n
\text{eval } (\text{Add } x \; y) = \text{case eval } x \text{ of }
\quad \text{Nothing} \rightarrow \text{Nothing }
\quad \text{Just } n \rightarrow \text{case eval } y \text{ of }
\quad \text{Nothing} \rightarrow \text{Nothing }
\quad \text{Just } m \rightarrow \text{Just } (n + m)
\text{eval } (\text{Throw}) = \text{Nothing }
\text{eval } (\text{Catch } x \; y) = \text{case eval } x \text{ of }
\quad \text{Nothing} \rightarrow \text{eval } y
\quad \text{Just } n \rightarrow \text{Just } n
$$

We will now calculate an abstract machine from this extended semantics, by following the same three-step process as previously. That is, we first add continuations, then defunctionalize, and finally refactor the definitions.
Step 1: Add continuations

Because our semantics now returns a result of type \( \text{Maybe Int} \), the type of continuations that we use must be modified accordingly:

\[
\text{type } \text{Cont} \ = \ \text{Maybe Int} \rightarrow \text{Maybe Int}
\]

Our aim now is to define a new semantics

\[
eval' :: \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Maybe Int}
\]

such that:

\[
eval' \ e \ c \ = \ c \ (\eval \ e)
\]

That is, the new semantics behaves in the same way as \( \eval \), except that it applies a continuation to the result. As previously, we can calculate a recursive definition for \( \eval' \) directly from this equation by structural induction on \( \text{Expr} \):

\[
\begin{align*}
\eval' \ &:: \ \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Maybe Int} \\
\eval' \ (\text{Val } n) \ c &\ = \ c \ (\text{Just } n) \\
\eval' \ (\text{Throw}) \ c &\ = \ c \ \text{Nothing} \\
\eval' \ (\text{Add } x \ y) \ c &\ = \ \eval' x \ (\lambda x' \rightarrow \text{case } x' \ of \text{Nothing} \rightarrow c \ \text{Nothing} \\
&\qquad \text{Just } n \rightarrow \eval' y \ (\lambda y' \rightarrow \text{case } y' \ of \text{Nothing} \rightarrow c \ \text{Nothing} \\
&\qquad \text{Just } m \rightarrow c \ (\text{Just } (n+m))) \\
\eval' \ (\text{Catch } x \ y) \ c &\ = \ \eval' x \ (\lambda x' \rightarrow \text{case } x' \ of \text{Nothing} \rightarrow \eval' y \ c \\
&\qquad \text{Just } n \rightarrow c \ (\text{Just } n))
\end{align*}
\]

(The above and subsequent omitted calculations are included in the extended version of the article.) In turn, our original semantics can be recovered by invoking our new semantics with the identity continuation. That is, we have

\[
\begin{align*}
\eval &:: \ \text{Expr} \rightarrow \text{Maybe Int} \\
\eval \ e &\ = \ \eval' \ e \ (\lambda x \rightarrow x)
\end{align*}
\]

Step 2: Defunctionalize

Our new semantics uses four forms of continuations, namely one to invoke the semantics, two in the case for addition, and one in the case for catch. We define
four combinators for constructing these continuations:

\[
\begin{align*}
c1 &:: Cont \\
c1 &= \lambda x \rightarrow x \\
c2 &:: Expr \rightarrow Cont \rightarrow Cont \\
c2 y c &= \lambda x' \rightarrow \text{case } x' \text{ of} \\
&\quad \text{Nothing } \rightarrow c \text{ Nothing} \\
&\quad \text{Just } n \rightarrow \text{eval'} y (c3 n c) \\
c3 &:: Int \rightarrow Cont \rightarrow Cont \\
c3 n c &= \lambda x' \rightarrow \text{case } x' \text{ of} \\
&\quad \text{Nothing } \rightarrow c \text{ Nothing} \\
&\quad \text{Just } m \rightarrow c \text{ (Just } (n + m)) \\
c4 &:: Expr \rightarrow Cont \rightarrow Cont \\
c4 y c &= \lambda x' \rightarrow \text{case } x' \text{ of} \\
&\quad \text{Nothing } \rightarrow \text{eval'} y c \\
&\quad \text{Just } n \rightarrow c \text{ (Just } n) \\
\end{align*}
\]

Using these combinators, our semantics can now be rewritten as follows:

\[
\begin{align*}
eval' &:: Expr \rightarrow Cont \rightarrow Maybe Int \\
eval' (\text{Val } n) c &= c \text{ (Just } n) \\
eval' (\text{Throw}) c &= c \text{ Nothing} \\
eval' (\text{Add } x y) c &= \text{eval'} x (c2 y c) \\
eval' (\text{Catch } x y) c &= \text{eval'} x (c4 y c) \\
eval e &= Expr \rightarrow Maybe Int \\
eval e c1 &= \text{eval'} e c1 \\
\end{align*}
\]

We now define a datatype to represent the four combinators, together with an application function that formalises the representation:

\[
\begin{align*}
data \text{ CONT} &= C1 \mid C2 \text{ Expr CONT} \mid C3 \text{ Int Cont} \mid C4 \text{ Expr CONT} \\
apply &:: \text{ CONT} \rightarrow Cont \\
apply C1 &= c1 \\
apply (C2 y c) &= c2 y (apply c) \\
apply (C3 n c) &= c3 n (apply c) \\
apply (C4 y c) &= c4 y (apply c) \\
\end{align*}
\]

Our aim now is to define a new semantics

\[
eval'' :: Expr \rightarrow \text{ CONT} \rightarrow Maybe Int \\
\]

such that:

\[
eval'' e c = \text{eval'} e (\text{apply } c) \\
\]

That is, the new semantics behaves in the same way as \text{eval'}, except that it uses representations of continuations rather than actual continuations. We can calculate
the definition for \( \text{eval''} \) by structural induction on \( \text{Expr} \):

\[
\begin{align*}
\text{eval''} & :: \text{Expr} \to \text{CONT} \to \text{Maybe Int} \\
\text{eval''} \ (\text{Val} \ n) \ c &= \ \text{apply} \ c \ (\text{Just} \ n) \\
\text{eval''} \ (\text{Throw}) \ c &= \ \text{apply} \ c \ \text{Nothing} \\
\text{eval''} \ (\text{Add} \ x \ y) \ c &= \ \text{eval''} \ x \ (\text{C2} \ y \ c) \\
\text{eval''} \ (\text{Catch} \ x \ y) \ c &= \ \text{eval''} \ x \ (\text{C4} \ y \ c)
\end{align*}
\]

In turn, we can calculate a new definition for \( \text{apply} \) by case analysis:

\[
\begin{align*}
\text{apply} & :: \text{CONT} \to \text{Maybe Int} \to \text{Maybe Int} \\
\text{apply} \ C1 \ x &= x \\
\text{apply} \ (\text{C2} \ y \ c) \ \text{Nothing} &= \ \text{apply} \ c \ \text{Nothing} \\
\text{apply} \ (\text{C2} \ y \ c) \ (\text{Just} \ n) &= \ \text{eval''} \ y \ (\text{C3} \ n \ c) \\
\text{apply} \ (\text{C3} \ n \ c) \ \text{Nothing} &= \ \text{apply} \ c \ \text{Nothing} \\
\text{apply} \ (\text{C3} \ n \ c) \ (\text{Just} \ m) &= \ \text{apply} \ c \ (\text{Just} \ (n + m)) \\
\text{apply} \ (\text{C4} \ y \ c) \ \text{Nothing} &= \ \text{eval''} \ y \ c \\
\text{apply} \ (\text{C4} \ y \ c) \ (\text{Just} \ n) &= \ \text{apply} \ c \ (\text{Just} \ n)
\end{align*}
\]

Our original semantics can be recovered by invoking our new semantics with the representation of the identity continuation:

\[
\begin{align*}
\text{eval} & :: \text{Expr} \to \text{Maybe Int} \\
\text{eval} &= \ \text{eval''} \ e \ C1
\end{align*}
\]

**Step 3: Refactor**

We now rename the components in the same way as previously, and rename the new combinator \( \text{C4} \) as \( \text{HAND} \). This time around, however, refactoring amounts to more than just renaming. In particular, we split the application function

\[
\text{apply} :: \text{Cont} \to \text{Maybe Int} \to \text{Maybe Int}
\]

into two separate application functions

\[
\begin{align*}
\text{exec} & :: \text{Cont} \to \text{Int} \to \text{Maybe Int} \\
\text{unwind} & :: \text{Cont} \to \text{Maybe Int}
\end{align*}
\]

such that:

\[
\begin{align*}
\text{apply} \ c \ (\text{Just} \ n) &= \ \text{exec} \ c \ n \\
\text{apply} \ c \ \text{Nothing} &= \ \text{unwind} \ c
\end{align*}
\]

That is, \( \text{exec} \) deals with normal arguments, and \( \text{unwind} \) with exceptional arguments. We can calculate the definitions for \( \text{exec} \) and \( \text{unwind} \) by structural induc-
tion on Cont, as a result of which we obtain the following machine:

\[
\text{data } \text{Cont} = \text{STOP} | \text{EVAL } \text{Expr } \text{Cont} | \text{ADD } \text{Int } \text{Cont} | \text{HAND } \text{Expr } \text{Cont}
\]

\[
\text{eval} :: \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Maybe } \text{Int}
\]

\[
\text{eval } (\text{Val } n) \ c = \text{exec } c \ n
\]

\[
\text{eval } (\text{Throw}) \ c = \text{unwind } c
\]

\[
\text{eval } (\text{Add } x \ y) \ c = \text{eval } x \ (\text{EVAL } y \ c)
\]

\[
\text{eval } (\text{Catch } x \ y) \ c = \text{eval } x \ (\text{HAND } y \ c)
\]

\[
\text{exec} :: \text{Cont} \rightarrow \text{Int} \rightarrow \text{Maybe } \text{Int}
\]

\[
\text{exec } \text{STOP } n = \text{Just } n
\]

\[
\text{exec } (\text{EVAL } y \ c) \ n = \text{eval } y \ (\text{ADD } n \ c)
\]

\[
\text{exec } (\text{ADD } n \ c) \ m = \text{exec } c \ (n + m)
\]

\[
\text{exec } (\text{HAND } y \ c) \ n = \text{exec } c \ n
\]

\[
\text{unwind} :: \text{Cont} \rightarrow \text{Maybe } \text{Int}
\]

\[
\text{unwind } \text{STOP} = \text{Nothing}
\]

\[
\text{unwind } (\text{EVAL } c) = \text{unwind } c
\]

\[
\text{unwind } (\text{ADD } c) = \text{unwind } c
\]

\[
\text{unwind } (\text{HAND } y \ c) = \text{eval } y \ c
\]

\[
\text{run} :: \text{Expr} \rightarrow \text{Maybe } \text{Int}
\]

\[
\text{run } e = \text{eval } e \ \text{STOP}
\]

We now explain the three main functions of the abstract machine:

- **eval** evaluates an expression in the context of a control stack. The cases for integer values and addition are as previously. If the expression is a throw, we **unwind the stack** seeking a handler expression. If the expression is a catch, we evaluate its first argument, \(x\), and **mark the stack** with the instruction \(\text{HAND } y\) to indicate that its second argument, the handler \(y\), should be used if evaluation of its first produces an exceptional value.

- **exec** executes a control stack in the context of an integer argument. The first three cases are as previously, except that if the stack is empty the resulting integer is tagged as a normal result value. If the top of the stack is a handler instruction, there is no need for the associated handler expression because a normal integer result has already been produced, and we **unmark the stack** by popping the handler and then continue executing.

- **unwind** executes the control stack in the context of an exception. If the stack is empty, the exception is uncaught and we simply return the exceptional result value. If the top of the stack is an evaluation or an addition instruction, there is no need for their arguments because a handler is being sought, and we pop them from the stack and then continue unwinding. If the top of the stack is a handler instruction, we catch the exception by evaluating the associated handler expression in the context of the remaining stack.
Chapter 4. Calculating an Exceptional Machine

Note that the idea of marking, unmarking, and unwinding the stack arose directly from the calculations, and did not require any prior knowledge of these concepts. It is also interesting to note that the above machine produced by calculation is both simpler and more efficient than those we had previously designed by hand. In particular, our previous machines did not make a clean separation between the three concepts of evaluating an expression (eval), executing the control stack (exec) and unwinding the control stack (unwind).

To illustrate our machine, here is how it evaluates $1 + (catch (2 + throw) 3)$:

\[
\begin{align*}
\text{run} & \left( Add(Val 1) \left( Catch(Add(Val 2)\ Throw)(Val 3) \right) \right) \\
= & \ \text{eval} \left( Add(Val 1) \left( Catch(Add(Val 2)\ Throw)(Val 3) \right) \right) \ STOP \\
= & \ \text{eval} \left( Val 1 \right) \left( \text{EVAL} \left( Add(Val 2)\ Throw\right)(Val 3) \right) \ STOP \\
= & \ \text{exec} \left( \text{EVAL} \left( Catch(Add(Val 2)\ Throw)(Val 3) \right) \ STOP \right) \ 1 \\
= & \ \text{eval} \left( Add(Val 2)\ Throw\right)(Val 3) \ ADD \ 1 \ STOP \\
= & \ \text{exec} \left( \text{EVAL} \left( Add(Val 2)\ Throw\right)(Val 3) \ ADD \ 1 \ STOP \right) \\
= & \ \text{eval} \left( Val 2 \right) \left( \text{EVAL} \left( HAND\ (Val 3) \ ADD \ 1 \ STOP \right)\right) \\
= & \ \text{exec} \left( \text{EVAL} \left( HAND\ (Val 3) \ ADD \ 1 \ STOP \right)\right) \ 2 \\
= & \ \text{eval} \left( HAND\ (Val 3) \ ADD \ 1 \ STOP \right) \\
= & \ \text{exec} \left( HAND\ (Val 3) \ ADD \ 1 \ STOP \right) \ 3 \\
= & \ \text{exec} \ STOP \ 4 \\
= & \ 4
\end{align*}
\]

That is, the machine first proceeds normally by transferring control back and forward between the functions eval and exec, until the exception is encountered, at which point the control stack is unwound to find the handler expression, and the machine then proceeds normally once again.

### 4.5 FURTHER WORK

We have shown how an abstract machine for a small language with exceptions can be calculated in a systematic way from a semantics for the language, using a three-step process of adding continuations, defunctionalizing, and refactoring. Moreover, the calculations themselves are straightforward, only requiring the basic concepts of structural induction and case analysis.

Possible directions for further work include exploring the impact of higher-level algebraic methods (such as monads [Wad92] and folds [Hut99]) on the calculations, mechanically checking the calculations using a theorem proving system (for example, see [Nip04]), factorising the abstract machine into the composition of a compiler and a virtual machine [ABDM03a], and generalising the underlying language (we are particularly interested in the addition of interrupts).

**Acknowledgements.** Thanks to the referees, Thorsten Altenkirch, Olivier Danvy, Conor McBride, and the WG2.1 meeting in Nottingham for useful comments.
REFERENCES


Chapter 5

Generalizing the AUGMENT Combinator

Neil Ghani1, Tarmo Uustalu2, and Varmo Vene3

Abstract: The usual initial algebra semantics of inductive types provides a clear and uniform explanation for the FOLD combinator. In an APLAS 2004 paper [GUV04], we described an alternative equivalent semantics of inductive types as limits of algebra structure forgetting functors. This gave us an elegant universal property based account of the BUILD and AUGMENT combinators, which form the core of the shortcut deforestation program transformation method by Gill et al. [GLP93] [Gil96]. Here we present further evidence for the flexibility of our approach by showing that a useful AUGMENT-like combinator is definable for a far wider class of parameterized inductive types than free monads, namely for all monads arising from a parameterized monad via an initial algebra construction.

5.1 INTRODUCTION

The standard approach to programming with inductive types is based on the IN/FOLD (constructors/structural recursion) syntax derived directly from their initial algebra semantics. This encourages a modular style of programming where functions are composed of simpler functions using intermediate data structures as a glue. Apart from the obvious advantages for the programmer, the use of intermediate data structures, which have to be constructed, traversed and discarded, can make programs inefficient both in terms of time and space. This calls for methods for transforming a modular program into an equivalent but more efficient program where these intermediate structures have been eliminated.

1Dept. of Math. and Comp. Sci., University of Leicester, University Road, Leicester, LE1 7RH, UK; E-mail: N.Ghani@mcs.le.ac.uk
2Institute of Cybernetics, Tallinn Univ. of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia; E-mail: tarmo@cs.ioc.ee
3Dept. of Computer Science, University of Tartu, Liivi 2, EE-50409 Tartu, Estonia; E-mail: varmo@cs.ut.ee
One of the most successful methods for eliminating intermediate data structures has been shortcut deforestation, originally introduced by Gill et al. [GLP93, Gil96] for lists and generalized for other inductive types by Launchbury and Sheard and Takano and Meyer [LS95, TM95]. This is based on the interaction of FOLD with a novel constructor combinator BUILD, capturing uniform production of data structures in a fashion matching the consumption pattern of FOLD. Deforestation happens by repeated application of a simple transformation rule of FOLD/BUILD fusion. Gill [Gil96] also introduced an enhanced AUGMENT combinator for lists and Johann [Joh02] generalized it for arbitrary inductive types with distinguished non-recursive constructors (so that list types and their nil constructors became instances).

Despite the success of the shortcut deforestation method, its theoretical explanations (see for example [TM95]) have not been fully satisfactory, typically relying on informal uses of Wadler’s [Wad89] free theorems, without discussing the exact conditions under which these are available. It is only recently that Johann [Joh03, Joh02] has proved the correctness of FOLD/BUILD and FOLD/AUGMENT fusion via parametricity of contextual equivalence. In an APLAS 2004 paper [GUV04], we approached the issue from a different perspective. We set out to give a language-independent axiomatic specification of BUILD and AUGMENT and derived it from an elementary alternative semantics of inductive types as limits of algebra structure forgetting functors. That semantics has BUILD rather than IN as the basic constructor of an inductive type and explains also the impredicative encoding of inductive types.

This short paper is a companion paper to [GUV04]. We present further evidence for the merits of the transparency of our framework by deriving a strong generalization of the AUGMENT combinator. We observe that AUGMENT is a monadic operation and show that AUGMENT is definable not only for free monads, which is essentially what Johann’s version [Joh02] amounts to, but far more generally for any monad arising from a parameterized monad via the initial algebras based construction of Uustalu [Uus03]. The extension operation of such monads can be seen as a grafting operation. The FOLD/AUGMENT fusion rule makes it possible to deforest programs involving this form of grafting.

The paper is organized as follows. In Sections 2, 3 we give a very condensed motivation for BUILD and AUGMENT combinators and review the account of FOLD/BUILD and AUGMENT from [GUV04]. In Section 4, which is the main section, we extend this account to the generalized version AUGMENT, to conclude in Section 5.

We use Haskell in our examples, because its syntax is familiar to many, but this is purely illustrative. We are by no means asserting that Haskell’s polymorphism is fully compliant with the parametricity requirements imposed by our semantics.

### 5.2 Semantics of Fold/Build

Traditionally, the inductive type determined by an endofunctor $F$ on a semantic category $\mathcal{C}$ is modelled by a chosen initial $F$-algebra. Such a characterization via
a universal property immediately suggests a syntax and a theory for programming with the type in question and for reasoning about it. The initial algebra semantics gives us the familiar IN/FOLD syntax and theory, see for example [Hag87]. Indeed, by introducing the notations $(\mu F, \text{in}_F)$ for the initial $F$-algebra and $\text{fold}_{F,X,\phi}$ for the unique map to an $F$-algebra $(X, \phi)$, we institute a type $\mu F$ (the inductive type) equipped with a constructor $\text{in}_F$ and a destructor $\text{fold}_F$ (the corresponding structural recursor) with the typing rules

$$
\frac{(X, \phi) \in F\text{-alg}}{\text{in}_F : F(\mu F) \rightarrow \mu F} \\
\frac{(X, \phi) \in F\text{-alg}}{\text{fold}_{F,X,\phi} : \mu F \rightarrow X}
$$

$\beta$-conversion rule

$$
\frac{(X, \phi) \in F\text{-alg}}{\text{fold}_{F,X,\phi} \circ \text{in}_F = \phi \circ F \text{fold}_{F,X,\phi}}
$$

and $\eta$- and permutative conversion rules

$$
\frac{f : (X, \phi) \rightarrow (Y, \psi) \in F\text{-alg}}{\text{fold}_{F,\mu F} \text{in}_F = \text{id}_{\mu F}} \\
\frac{f \circ \text{fold}_{F,X,\phi} = \text{fold}_{F,Y,\psi}}
$$

As a well-known example, lists over type $A$ arise as the initial algebra of the functor $1 + A \times -$. The Haskell Prelude introduces the (parameterized) type of lists and equips it with IN/FOLD syntax as follows, the $\beta$-rule defining the computational behavior of the FOLD combinator.

```haskell
data [a] = [] | a : [a]
foldr :: (a -> x -> x) -> x -> [a] -> x
foldr c n [] = n
foldr c n (a : as) = c a (foldr c n as)
```

In shortcut deforestation, however, one uses a different syntax: producers of lists must be presented as applications of a BUILD combinator

```haskell
build :: (forall x. (a -> x -> x) -> x -> [a] -> x)
build theta = theta (:) []
```

The merit is that applications of FOLD can be cancelled by those of BUILD as described by the so-called FOLD/BUILD fusion law

```haskell
foldr c n (build theta) == theta c n
```

In [QUV04], we motivated this different syntax by showing that the inductive type given by $F$ can alternatively be interpreted as a chosen limit of the functor $U_F : F\text{-alg} \rightarrow C$. By definition, a $U_F$-cone is an object $C$ in $\mathcal{C}$ and, for any $F$-algebra $(X, \phi)$, a map $\Theta_X \phi : C \rightarrow X$ in $\mathcal{C}$, such that (*) for any $F$-algebra map $f : (X, \phi) \rightarrow (Y, \psi)$, we have $f \circ \Theta_X \phi = \Theta_Y \psi$. A $U_F$-cone map $h : (C, \Theta) \rightarrow (D, \Xi)$ is a map $h : C \rightarrow D$ in $\mathcal{C}$ such that, for any $F$-algebra $(X, \phi)$, we have $\Xi_X \phi \circ h = \Theta_X \phi$. A $U_F$-limit is a $U_F$-cone to which there is a unique map from any other
$U_F$-cone. Writing $(\mu^*F, \text{fold}_F^*)$ for the $U_F$-limit (in case it exists) and $\text{build}_F^*C$ for the unique map from a $U_F$-cone $(C, \Theta)$, we obtain a type $\mu^*F$ with a destructor $\text{fold}_F^*$ and a constructor $\text{build}_F^*$ governed by the typing rules

\[
\frac{(X, \varphi) \in F\text{-alg}}{\text{fold}_F^*X\varphi : \mu^*F \to X} \quad \frac{f : (X, \varphi) \to (Y, \psi) \in F\text{-alg}}{f \circ \text{fold}_F^*X\varphi = \text{fold}_F^*Y\psi} \quad \frac{(C, \Theta) \in U_F\text{-cone}}{\text{build}_F^*C \Theta : C \to \mu^*F}
\]

The corresponding $\beta$-conversion rule is

\[
\frac{(C, \Theta) \in U_F\text{-cone} \quad (X, \varphi) \in F\text{-alg}}{\text{fold}_F^*X\varphi \circ \text{build}_F^*C \Theta = \Theta_X \varphi}
\]

and the $\eta$- and permutative conversion rules are

\[
\text{id}_{\mu^*F} = \text{build}_F^*\mu^*F \text{fold}_F^* \quad \frac{h : (C, \Theta) \to (D, \Xi) \in U_F\text{-cone}}{\text{build}_F^*D \Xi \circ h = \text{build}_F^*C \Theta}
\]

Inspecting these rules, one sees immediately that $\text{fold}_F^*$ and $\text{build}_F^*$ type exactly as $\text{FOLD}$ and $\text{BUILD}$ should, although with the reservation that the argument of $\text{build}_F^*$ is required to meet the coherence condition (*). Moreover, the $\beta$-conversion rule is axiomatic $\text{FOLD}/\text{BUILD}$ fusion. This makes the limit of forgetful functor semantics an appealing candidate justification for the $\text{FOLD}/\text{BUILD}$ syntax (with $\text{BUILD}$ rather than $\text{IN}$ as the primitive constructor) and for the fusion rule. Remarkably, the condition (*) crisply explicates the kind of parametricity of polymorphism that is required for everything to work.

That the alternative semantics is adequate by being equivalent to the standard one is demonstrated by the following proposition.

**Proposition 5.1.** Let $C$ be a category and $F : C \to C$ be a functor. (a) If there is an initial $F$-algebra $(\mu F, \text{in}_F)$, then $\mu F$ is the vertex of a $U_F$-limit. (b) If there is a $U_F$-limit $(\mu^*F, \text{fold}_F^*)$, then $\mu^*F$ is the carrier of an initial $F$-algebra.

**Proof (constructions).** (a) Set, for any $U_F$-cone $(C, \Theta)$, $\text{build}_F^*C \Theta = \text{df} \Theta_{\mu F} \text{in}_F$; $(\mu F, \text{fold}_F^*)$ is a $U_F$-cone and $\text{build}_F^*C \Theta$ a unique map from any $U_F$-cone.

(b) Set, for any $F$-algebra $(X, \varphi)$, $\text{infol}(X, \varphi) = \text{df} \varphi \circ F \text{fold}_F^*X\varphi$; set $\text{in}_F^* = \text{df} \text{build}_F^*F(\mu F) \text{infol}^*$; $(\mu^*F, \text{in}_F^*)$ is an $F$-algebra and $\text{fold}_F^*X\varphi$ is a unique map to any $F$-algebra $(X, \varphi)$.

The condition (*) turns out to be equivalent to a strong dinaturality condition.

**Definition 5.1.** Let $H, K : C^\text{op} \times C \to \mathcal{D}$ be functors. A strongly dinatural transformation $\Theta : H \to K$ is a family of maps $\Theta_X : H(X, X) \to K(X, X)$ in $\mathcal{D}$ for all objects $X$ in $C$ such that, for every map $f : X \to Y$ in $C$, object $W$ and maps $p_0 : W \to H(X, X)$, $p_1 : W \to H(Y, Y)$ in $\mathcal{D}$, if the square in the following diagram
Proposition 5.2. Let $\mathcal{C}$ be a locally small category and $F : \mathcal{C} \to \mathcal{C}$ a functor. A $U_F$-cone structure with vertex $C$ is the same thing as a strong dinatural transformation from $\text{Hom}(F -, -)$ to $\text{Hom}(C, -)$.

The relevance of this proposition for programming languages is that the coherence condition (*) is for free whenever strong dinaturality is.

5.3 THE AUGMENT OF FREE MONADS

BUILD presentations of producers do not always suffice for shortcut deforestation. A prototypical example for lists is the append function

```haskell
append as bs = foldr (:) bs as
```

The definition as a BUILD

```haskell
append as bs = build (\ c n -> foldr c (foldr c n bs) as)
```

introduces an unnatural traversal of the list appended and this traversal is not necessarily removed in shortcut deforestation.

The solution proposed by Gill [Gil96] consists in deploying an enhanced version of BUILD, called AUGMENT. For lists, the Haskell version of this combinator is the following:

```haskell
augment :: (forall x. (a -> x -> x) -> x -> x) -> [a] -> [a]
augment theta bs = theta (:) bs
```

An AUGMENT represents nothing else than a BUILD followed by an append:

```haskell
augment theta bs == append (build theta) bs
```

And conversely, a BUILD is just an AUGMENT applied to nil:

```haskell
build theta == augment theta []
```

But the reason to introduce AUGMENT is that it obeys a special law of fusion with FOLD:

```haskell
foldr c n (augment theta bs) == theta c (foldr c n bs)
```
This law is similar by its spirit to the FOLD/BUILD law. But differently from a BUILD, an AUGMENT does not cancel a FOLD completely: the reduct contains a residual FOLD.

The definition of the append function as an AUGMENT is

\[
\text{append as } \text{bs} = \text{augment } (\ \text{c n }\to\ \text{foldr c n as}) \text{ bs}
\]

A similar AUGMENT combinator can be easily defined for arbitrary inductive types with non-recursive constructors to obtain a version of shortcut deforestation tailored specifically for producer functions that perform grafting, see Johann [Joh02]. In [GUV04], we justified a slightly more general version in the style our FOLD/BUILD account and, in line with our general methodology, derived it from a unique existence situation. In that version, AUGMENT combinators are associated to free monads, which are families of inductive types rather than single inductive types.

Recall that a monad (in extension form) on a category \( \mathcal{C} \) is an object mapping \( T : |\mathcal{C}| \to |\mathcal{C}| \) together with, for any object \( A \), a map \( \eta_A : A \to TA (\text{unit}) \), and for any map \( f : A \to TB \), a map \( f^* : TA \to TB \) (extension operation) such that (i) for any map \( f : A \to TB \), \( f^* \circ \eta_A = f \), (ii) for any object \( A \), \( \eta_A^* = \text{id}_{TA} \), and (iii) for any maps \( f : A \to TB \), \( g : B \to TC \), \( (g^* \circ f)^* = g^* \circ f^* \).

Recall also that monads in Haskell are implemented via a type constructor class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

The Haskell jargon for the extension operation is ‘bind’.

Consider an endofunctor \( H \) on a category \( \mathcal{C} \) and let \( T' : \mathcal{C} \to \left[ \mathcal{C}, \mathcal{C} \right] \) be the functor given by \( T'AX = A + HX \). Then, if an initial \( T'A \)-algebra (= a free \( H \)-algebra over \( A \)) exists for every object \( A \) of \( \mathcal{C} \), we can get a functor \( T : \mathcal{C} \to \mathcal{C} \) by defining \( TA = \mu(T'A) \). This models an inductive type parameterized in non-recursive constructors. Decompose each map \( \text{in}_{T'A} \) into two maps \( \eta_A : A \to TA \) and \( \tau_A : H(TA) \to TA \) by setting

\[
\eta_A = \text{df } \text{in}_{T'A} \circ \text{in}_{A,H(TA)} \\
\tau_A = \text{df } \text{in}_{T'A} \circ \text{in}_{A,H(TA)}
\]

Define, finally, for any map \( f : A \to TB \), a map \( f^* : TA \to TB \) by

\[
f^* = \text{df } \text{fold}_{T'A,TB}[f, \tau_B]
\]

Conceptually, \( \eta \) packages the non-recursive constructors of the parameterized type, \( \tau \) packages the recursive constructors, and \( ()^* \) is substitution for non-recursive constructors.

It is standard knowledge the data \( (T, \eta, ()^*) \) so constructed constitute a monad which, more specifically, is the free monad generated by the functor \( H \).

The following proposition supplies the parameterized inductive type \( T \) with an AUGMENT combinator.
Chapter 5. Generalizing the AUGMENT Combinator

**Proposition 5.3.** Let $C$ be a category, $H : C \to C$ a functor such that initial algebras of all functors $T'/A =_A A + H$ exist. Let $T$, $\eta$, $\tau$, $()^*$ be defined as above. Then, for any map $f : A \to TB$ and $U_{T'A}$-cone $(C, \Theta)$ there exists a unique map $h : C \to TB$ such that, for any $T'B$-algebra $(X, [\varphi_0, \varphi_1])$, it holds that

$$\text{fold}_{T'B=X}[\varphi_0, \varphi_1] \circ h = \Theta_X([\text{fold}_{T'B=X}[\varphi_0, \varphi_1] \circ f, \varphi_1])$$

We denote the unique map by $\text{augment}_{T', C}(\Theta, f)$.

On the level of syntax, the proposition justifies the introduction of a constructor combinator $\text{augment}_{T'}$ with a typing rule

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone} \quad f : A \to TB}{\text{augment}_{T', C}(\Theta, f) : C \to TB}$$

and $\beta$-conversion rule

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone} \quad f : A \to TB \quad (X, [\varphi_0, \varphi_1]) \in T'B\text{-alg}}{\text{fold}_{T'B=X}[\varphi_0, \varphi_1] \circ \text{augment}_{T', C}(\Theta, f) = \Theta_X[\text{fold}_{T'B=X}[\varphi_0, \varphi_1] \circ f, \varphi_1]}$$

which is fusion of $\text{FOLD}$ and $\text{AUGMENT}$. One also gets a number of conversion rules pertaining to the monad structure on $T$, some of which we will list in the next section.

The combinator of Johann [Joh02] differs from our version by having the type of non-recursive constructors fixed (so that $B = A$), thus confining the action of the combinator to a single inductive type rather than a family one would obtain by letting the type of non-recursive constructors to vary. (The reason must be that, in the prototypical case of lists, the type of non-recursive constructors is constantly 1; normally, one does not consider the possibility of supporting multiple nil’s drawn from a parameter type.) Unfortunately, this restriction hides the important role that the monad structure on $T$ plays in the construction. This role is central for our generalization in the next section.

A typical example of a free monad is the parameterized type of binary leaf labelled trees. Binary leaf labelled trees with leaf label type $A$ are the initial algebra of the functor $A + x - x$. They are Haskell-implementable as follows:

```haskell
data BLTree a = Leaf a | Bin (BLTree a) (BLTree a)

foldB :: (a -> x) -> (x -> x -> x) -> BLTree a -> x
foldB l b (Leaf a) = l a
foldB l b (Bin as0 as1) = b (foldB l b as0) (foldB l b as1)

instance Monad BLTree where
    return a = Leaf a
    as >>= f = foldB f Bin as
```

The $\text{BUILD}$ and $\text{AUGMENT}$ combinators are implementable as follows:

```haskell
buildB :: (forall x. (a -> x) -> (x -> x -> x) -> BLTree a)
```
buildB theta = theta Leaf Bin

augmentB :: (forall x. (a -> x) -> (x -> x -> x) -> x)
  -> (a -> BLTree b) -> BLTree b
augmentB theta f = theta f Bin

The shortcut deforestation laws say that

\[
\text{foldB } l \ b \ (\text{buildB } \theta) = \theta \ l \ b
\]

\[
\text{foldB } l \ b \ (\text{augmentB } \theta \ f) = \theta \ (\text{foldB } l \ b . \ f) \ b
\]

It is also possible to give a generic implementation of the AUGMENT combinator of free monads that caters for all possible shapes of leaf-labelled trees at one go. The code is nearly identical to what we have just seen.

We observe that AUGMENT combinators are about uniform production of inductive data in combination with grafting. Grafting presupposes monadic structure. We have seen that an AUGMENT combinator can be defined for any free monad. But there exist parameterized inductive types carrying a non-free monad structure! Could it be that an AUGMENT-like combinator subject to a fusion law is definable for some of them too? We will answer this question affirmatively in the next section.

### 5.4 A GENERALIZED AUGMENT COMBINATOR

The AUGMENT combinator for the monad of the free algebras of a functor (or, in other words, the free monad) is a meaningful combinator obeying a useful FOLD/AUGMENT fusion rule, but it is not as general as possible. As we will now show, a similar combinator is possible for any monad obtained from a parameterized monad via initial algebras as described by Uustalu [Uus03]. This generalization is, to our knowledge, entirely new.

We must start by reviewing the construction in [Uus03].

A parameterized monad is an object mapping \( T' : |C| \rightarrow [|C|, |C|] \) together with, for any objects \( A, X \), a map \( \eta'_{A,X} : A \rightarrow T'AX \) and, for any map \( f : A \rightarrow T'BX \), a map \( f^\circ : T'AX \rightarrow T'BX \) such that \((i')\) for any map \( f : A \rightarrow T'BX \), \( f^\circ \circ \eta'_{A,X} = f \), \((ii')\) for any objects \( A, X \), \( (\eta'_{A,X})^\circ = \text{id}_{T'AX} \), \((iii')\) for any maps \( f : A \rightarrow T'BX \), \( g : B \rightarrow T'CX \), \( (g^\circ \circ f)^\circ = g^\circ \circ f^\circ \), \((iv')\) for any object \( A \) and map \( \xi : X \rightarrow Y \), \( T'A\xi \circ \eta'_{A,X} = \eta'_{A,Y} \), and \((v')\) for any maps \( f : A \rightarrow T'BX \), \( \xi : X \rightarrow Y \), \( T'B\xi \circ f^\circ = (T'B\xi \circ f)^\circ \circ T'A\xi \). It is easy to verify that a parameterized monad on \( C \) is essentially just a functor from \( C \) to the category \( \text{Monad}(C) \) of monads on \( C \), but we adopt the above presentation as this will be more convenient.

Now if \((T', \eta', (\circ))\) is a parameterized monad on \( C \), we can consider the initial algebras \((TA, \alpha_A) =_{df} (\mu(T'A), \text{int}_{T'A})\), provided all functors \( T'A \) have an initial algebra. Define, for any object \( A \), a map \( \eta_A : A \rightarrow TA \) by

\[
\eta_A =_{df} \alpha_A \circ \eta'_{A,TA}
\]
and, for any map \( f : A \to TB \), a map \( f^* : TA \to TB \) by

\[
f^* = \text{df} \ \text{fold}_{T'A, TB} (\alpha_B \circ (\alpha_B^{-1} \circ f))^\circ
\]

By a proposition in [Uus03], \((T, \eta, (\cdot)^\circ)\) is a monad. Its extension operation \((\cdot)^\circ\) is, in a meaningful but very liberal sense, a grafting operation.

Below are some examples of monads arising from parameterized monads in the manner just described:

- The monad of the free algebras of a given functor \( H : C \to C \) (the free monad generated by \( H \)). This is the example we already saw: the inducing parameterized monad \( T' \) is defined by \( T'AX = \text{df} A + HX \) (so that, for any type \( X \), \( T' - X \) is the exceptions monad with exceptions type \( HX \)).

- The monad of finitely branching node labelled trees (rose trees). The inducing parameterized monad \( T' \) is defined by \( T'AX = \text{df} A \times \text{List} X \). Notice that in this type of trees values are stored at inner nodes, not at leaves, so substitution must be a bit unusual. The substitution function \( f^* : TA \to TB \) that corresponds to a given substitution rule \( f : A \to TB \) is specified by the equation

\[
f^*(x[t_1, \ldots, t_n]) = \text{df} y[s_1, \ldots, s_m, f^*(t_1), \ldots, f^*(t_n)] \quad \text{where} \quad f(x) = \text{df} y[s_1, \ldots, s_m]
\]

- The monad of inductive hyperfunctions [KLP01] with a fixed domain. The inducing parameterized monad \( T' \) is defined by \( T'AX = \text{df} (X \Rightarrow E) \Rightarrow A \) where \( E \) is a fixed object of \( C \).

- A further interesting monad is obtained from the parameterized monad \( T' \) defined by \( T'AX = \text{df} \ \text{List}(A + X) \). The bind operation of this monad is defined in terms of the bind operation of the lists monad.

Now, as promised, we can introduce an \textsc{augment} combinator can for all monads that arise from a parameterized monad in the fashion described above. This results again from a uniquely satisfied property.

**Proposition 5.4.** Let \( C \) be a category and \((T', \eta', (\cdot)^\circ)\) a parameterized monad on it such that initial algebras of all functors \( T'A \) exist. Let \( T \eta, \tau, (\cdot)^\circ \) be defined as above. Then, for any map \( f : A \to TB \) and \( U_{T'A}\)-cone \((C, \Theta)\) there exists a unique map \( h : C \to TB \) such that, for any \( T'B \)-algebra \((X, \varphi)\), it holds that

\[
\text{fold}_{T'B, X} \varphi \circ h = \Theta_X (\varphi \circ (T'B (\text{fold}_{T'B, X} \varphi) \circ \alpha_B^{-1} \circ f))^\circ
\]

As earlier, we denote the unique map by \textsc{augment}_{\(T', C\)}(\(\Theta, f\)).

**Proof.** We prove that

\[
\text{augment}_{\(T', C\)}(\Theta, f) = \Theta_{TB} (\alpha_B \circ (\alpha_B^{-1} \circ f))^\circ
\]
For any $T'B$-algebra $(X, \varphi)$, making use of the axiom (v’) of parameterized monads, we have

$$\text{fold}_{T'B,X} \varphi \circ \alpha_B \circ (\alpha_B^{-1} \circ f)^\circ = \varphi \circ T'B (\text{fold}_{T'B,X} \varphi) \circ (\alpha_B^{-1} \circ f)^\circ = \varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha_B^{-1} \circ f)^\circ \circ T'A (\text{fold}_{T'B,X} \varphi)$$

Hence, $\text{fold}_{T'B,X} \varphi$ is a $T'A$-algebra map from $(TB, \alpha_B \circ (\alpha_B^{-1} \circ f)^\circ)$ to $(X, \varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha_B^{-1} \circ f)^\circ)$. Hence by $(C, \Theta)$ being a $U_{T'A}$-cone

$$\text{fold}_{T'B,X} \varphi \circ \Theta_{TB} (\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ) = \Theta_X (\varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha_B^{-1} \circ f)^\circ)$$

as needed.

Assume now we have a map $h : C \to TB$ such that, for any $T'B$-algebra $(X, \varphi)$,

$$\text{fold}_{T'B,X} \varphi \circ h = \Theta_X (\varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha_B^{-1} \circ f)^\circ)$$

Then

$$h = \text{fold}_{T'B,TB} \alpha_B \circ h = \Theta_{TB} (\alpha_B \circ (T'B (\text{fold}_{T'B,TB} \alpha_B) \circ \alpha_B^{-1} \circ f)^\circ) = \Theta_{TB} (\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ)$$

which completes the proof of uniqueness.

This unique existence motivates a combinator augment$_{T'}$ obeying the following rules:

- Typing rule:

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone}}{\text{augment}_{T',C}(\Theta, f) : C \to TB} f : A \to TB$$

- $\beta$-conversion rule (= FOLD/AUGMENT fusion law):

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone}}{\text{fold}_{T'B,X} \varphi \circ \text{augment}_{T',C}(\Theta, f) = \Theta_X (\varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha_B^{-1} \circ f)^\circ)} f : A \to TB \quad (X, \varphi) \in T'B\text{-alg}$$

- Conversion rules relating to the monad structure:

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone}}{\text{augment}_{T',C}(\Theta, f) = f^* \circ \text{build}_{T'A,C} \Theta} f : A \to TB$$

$$\frac{(C, \Theta) \in U_{T'A}\text{-cone}}{\text{build}_{T'A,C} \Theta = \text{augment}_{T',C}(\Theta, \eta_A)} (C, \Theta) \in U_{T'A}\text{-cone}$$

$$\frac{g^* \circ \text{augment}_{T',C}(\Theta, f) = \text{augment}_{T',C}(\Theta, g^* \circ f)}{g^* : A \to TC \quad g : B \to TC}$$
Chapter 5. Generalizing the \textsc{AUGMENT} Combinator

Notice that the first two rules in the last group state that the \textsc{AUGMENT} and \textsc{BUILD} combinators are interdefinable using the unit and bind of the monad.

Instantiating the first of them with \((C, \Theta) = (TA, \text{fold}_{T'A})\), we obtain that bind is \textsc{AUGMENT} applied to \textsc{FOLD}:

\[
f : A \to TB \\
\text{augment}_{T',TA}(\text{fold}_{T'A}; f) = f^*
\]

Combining this with the \textsc{FOLD}/\textsc{AUGMENT} fusion rule, we get the following rule for fusing \textsc{FOLD} with bind:

\[
f : A \to TB \quad (X, \varphi) \in T' B\text{-alg} \\
\text{fold}_{T'B,X} \circ f^* = \text{fold}_{T'A,X} (\varphi \circ (T'B (\text{fold}_{T'B,X} \varphi) \circ \alpha^{-1}_B \circ f^*))
\]

Here is an implementation of finitely branching node labelled trees (rose trees) in Haskell, including the monad structure.

```haskell
data Tree a = Node a [Tree a]

foldT :: (a -> [x] -> x) -> Tree a -> x
foldT phi (Node a tas) = phi a (map (foldT phi) tas)

instance Monad Tree where
  return a = Node a []
  ta >>= f = foldT (\a tbs' -> let Node b tbs = f a
                       in Node b (tbs ++ tbs')) ta

The \textsc{BUILD} and \textsc{AUGMENT} combinators are implementable as follows:

```haskell
buildT :: (forall x . (a -> [x] -> x) -> x) -> Tree a
buildT theta = theta Node

augmentT :: (forall x . (a -> [x] -> x) -> x) -> (a -> Tree b) -> Tree b
augmentT theta f = theta (\a tbs' -> let Node b tbs = f a
                                     in Node b (tbs ++ tbs'))
```

\textsc{FOLD}/\textsc{BUILD} fusion and \textsc{FOLD}/\textsc{AUGMENT} fusion say that

```
foldT phi (buildT theta) == theta phi
foldT phi (augmentT theta f) == theta (\a xs ->
  let Node b tbs = f a
      in phi b (map (foldT phi) tbs ++ xs))
```

For inductive hyperfunction spaces, we get the \textsc{BUILD} and \textsc{AUGMENT} combinators as follows:

```haskell
data Hyper e a = H { unH :: (Hyper e a -> e) -> a }
```
foldH :: (((x -> e) -> a) -> x) -> Hyper e a -> x
foldH phi (H h) = phi (\ g -> h (g . foldH phi))

instance Monad (Hyper e) where
  return a = H (const a)
  ha >>= f = foldH (\ g -> H (\ k -> unH (f (g k)) k)) ha

buildH :: (forall x . (((x -> e) -> a) -> x) -> x) -> Hyper e a
buildH theta = theta H

augmentH :: (forall x . (((x -> e) -> a) -> x) -> x) -> (a -> Hyper e b) -> Hyper e b
augmentH theta f = theta (\ g -> H (\ k -> unH (f (g k)) k))

The corresponding fusion laws are
foldH phi (buildH theta) == theta phi
foldH phi (augmentH theta f) ==
  theta (\ g -> phi (\ k -> unH (f (g k)) (k . foldH phi)))

Finally, the general construction of monads from parameterized monads via initial
algebras may also be implemented generically:

class Monad' m where
  return' :: a -> m a x
  (>>|) :: m a x -> (a -> m b x) -> m b x
  fmap' :: (x -> y) -> (m a x -> m a y)

data Mu m a = In {unIn :: m a (Mu m a) }

fold :: Monad' m m => (m a x -> x) -> Mu m a -> x
fold phi (In tas) = phi (fmap' (fold phi) tas)

instance Monad' m => Monad (Mu m) where
  return a = In (return' a)
  ta >>= f = fold (In . (>>| unIn . f)) ta

build :: (forall x . (m a x -> x) -> x) -> Mu m a
build theta = theta In

augment :: Monad' m => (forall x . (m a x -> x) -> x)
  -> (a -> Mu m b) -> Mu m b
augment theta f = theta (In . (>>| unIn . f))

The corresponding FOLD/BUILD and FOLD/AUGMENT fusion rules are as follows:
fold phi (build theta) == theta phi
fold phi (augment theta f) ==
  theta (phi . (>>| fmap' (fold phi) . unIn . f))
5.5 CONCLUSION AND FUTURE WORK

We have demonstrated that the semantics of inductive types in terms of limits of algebra structure forgetting functors does not only explain the typing (including, notably, the coherence condition on the argument) of the BUILD combinator and the FOLD/BUILD fusion rule, but exhibits a flexibility thanks to which it is not hard to generalize Gill’s originally rather specific AUGMENT combinator to a considerably wider class of parameterized inductive types. This reveals, in particular, that the original FOLD/AUGMENT fusion for lists, which at first sight looks like an ad hoc rule, is really a special case of a very systematic rule, a tip of a small iceberg.

As future work we plan to achieve a similar account for the vanish combinators of Voigtlander [Voi02]. We also plan to study the relation of strong dinaturality and parametricity.

Acknowledgments. The authors were partially supported by the Royal Society ESEP programme within joint research project No. 15642. The second and third author were also partially supported by the Estonian Science Foundation under grant No. 5567.

REFERENCES


Chapter 6

Alice Through the Looking Glass

Andreas Rossberg¹, Didier Le Botlan¹, Guido Tack¹, Thorsten Brun-klaus¹, Gert Smolka¹

Abstract: We present Alice, a functional programming language that has been designed with strong support for typed open programming. It incorporates concurrency with data flow synchronisation, higher-order modularity, dynamic modules, and type-safe pickling as a minimal and generic set of simple, orthogonal features providing that support. Based on these mechanisms Alice offers a flexible notion of component, and high-level facilities for distributed programming.

6.1 INTRODUCTION

Software is decreasingly delivered as a closed, monolithic whole. As its complexity and the need for integration grows it becomes more and more important to allow flexible dynamic acquisition of additional functionality. Program execution is no longer restricted to one local machine only: with net-oriented applications being omni-present, programs are increasingly distributed across local or global networks. As a result, programs need to exchange more data with more structure. In particular, they need to exchange behaviour, that is, data may include code.

We refer to development for the described scenario as open programming. Our understanding of open programming includes the following main characteristics:

- **Modularity**, to flexibly combine software blocks that were created separately.
- **Dynamicity**, to import and export software blocks in running programs.
- **Security**, to safely deal with unknown or untrusted software blocks.
- **Distribution**, to communicate data and software blocks over networks.
- **Concurrency**, to deal with asynchronous events and non-sequential tasks.

¹Programming Systems Lab, Saarland University, Saarbrücken, Germany; E-mail: rossberg,botlan,tack,brunklaus,smolka@ps.uni-sb.de
Dynamic software blocks are usually called components.

Most practical programming languages today have not been designed with open programming in mind. Even the few that have been – primarily Java [GJS96] – do not adequately address all of the above points. For example, Java is not statically type-safe, has only weak support for import/export, and rather clunky distribution and concurrency mechanisms.

Our claim is that only a few simple, orthogonal concepts are necessary to provide a flexible, strongly typed framework supporting all aspects of open programming. Components are a central notion in this framework, but instead of being primitive they can be derived from the following simpler, orthogonal concepts:

- **Futures**, which allow for light-weight concurrency and laziness.
- **Higher-order modules**, to provide genericity and encapsulation.
- **Packages**, to wrap modules into self-describing, dynamically typed entities.
- **Pickling**, to enable generic persistence and distribution.
- **Proxy functions**, to enable remote calls into other processes.

To substantiate our claim, we developed the programming language Alice ML, a conservative extension of Standard ML [MTHM97]. It is largely inspired by Oz [Smo95 Moz04 VH04], a relational language with advanced support for open programming, but lacking any notion of static typing. The aim of the Alice project is to systematically reconstruct the essential functionality of Oz on top of a typed functional language.

Alice has been implemented in the Alice Programming System [Ali04], a full-featured programming environment based on a VM with just-in-time compilation, support for platform-independent persistence and platform-independent mobile code, and a rich library for constraint programming.

**Organisation of the paper.** This paper describes the design of the open programming features of Alice. Futures provide for concurrency (Section 6.2). Higher-order modules enhance modularity (Section 6.3). Type-safe marshalling is enabled by packages (Section 6.4). Components (Section 6.5) are built on top of these mechanism (Section 6.6). Distribution is based on components and RPCs (Section 6.7). We briefly discuss the implementation (Section 6.8), compare to some related work (Section 6.9), and conclude with a short outlook (Section 6.10).

### 6.2 FUTURES

Programs communicating with the outside world usually have to deal with non-deterministic, asynchronous events. Purely sequential programming cannot adequately handle such scenarios. Support for concurrency hence is vital.

Concurrency in Alice is based uniformly on the concept of futures, which has been mostly adapted from Multilisp [Hal85]. A future is a transparent placeholder for a yet undetermined value that allows for implicit synchronisation based on data flow. There are different kinds of futures, which we will describe in the following sections. A formal semantics can be found in [NSS02]. Futures are a
generic mechanism for communication and synchronisation. As such, they are comparatively simple, but expressive enough to enable formulation of a broad range of concurrency abstractions.

### 6.2.1 Concurrency

Future-based concurrency is very light-weight: any expression can be evaluated in its own thread, a simple keyword allows forking off a concurrent computation:

```
spawn exp
```

This phrase immediately evaluates to a fresh *concurrent future*, standing for the yet unknown result of `exp`. Simultaneously, evaluation of `exp` is initiated in a new thread. As soon as the thread terminates, the result globally replaces the future.

A thread is said to *touch* a future when it performs an operation that requires the actual value the future stands for. A thread that touches a future is suspended automatically until the actual value is determined. This is known as *data flow synchronisation*.

If a concurrent thread terminates with an exception, the respective future is said to be *failed*. Any operation touching a failed future will cause the respective exception to be synchronously re-raised in the current thread.

Thanks to futures, threads give results, and concurrency can be orthogonally introduced for arbitrary parts of an expression. We hence speak of *functional threads*. For example, to evaluate all constituents of the application `e_1(e_2, e_3)` concurrently, it is sufficient to annotate the application as follows:

```
(spawn e_1) (spawn e_2, spawn e_3)
```

Functional threads allow turning a synchronous function call to a function `f` into an *asynchronous* one by simply prefixing the application with `spawn`:

```
val result = spawn f (x, y, z)
```

The ease of making asynchronous calls even where a result is required is important in combination with distributed programming (Section 6.7), because it allows for *lag tolerance*: the caller can continue its computation while waiting for the result to be delivered. Data flow synchronisation ensures that it will wait if necessary, but at the latest possible time, thus maximising concurrency.

Futures already provide for complex communication and synchronisation. Consider the following example:

```
val offset = spawn (sleep(Time.fromSeconds 120); 20)
val table = Vector.tabulate (40, fn i => spawn fib(i + offset))
```

The first declaration starts a thread that takes two minutes to deliver the value 20. The computation for the table entries in the second declaration depends on that value, but since the entries are computed concurrently, construction of the table can proceed without delay. However, the spawned threads will all block until `offset` is determined. Consecutive code can already access the table, but if it touches an entry that is not yet determined, it will automatically block.
Besides implicit synchronisation, Alice offers primitives for explicit synchronisation, including non-deterministic choice. They are sufficient to encode complex synchronisation with multiple events or timeouts [RLT+04].

### 6.2.2 Laziness

It has become a common desire to marry eager and lazy evaluation, and the future mechanism provides an elegant way to do so. While keeping eager application semantics, full support for laziness is available through *lazy futures*: the phrase

```
lazy exp
```

will not evaluate `exp`, but instead returns a fresh lazy future, standing for the yet unknown result of `exp`. Evaluation of `exp` is triggered by a thread first touching the future. At that moment, the lazy future becomes a concurrent future and evaluation proceeds as for concurrent futures.

In other words, lazy evaluation can be selected for individual expressions. For example, the expression `(fn x ⇒ 5) (lazy raise E)` will *not* raise `E`. A fully lazy evaluation regime can be emulated by prefixing *every* subexpression with `lazy`.

### 6.2.3 Promises and locking

Functional threads and lazy evaluation offer convenient means to introduce and eliminate futures. However, the direct coupling between a future and the computation delivering its value often is too inflexible. *Promises* are a more fine-grained mechanism that allows for creation and elimination of futures in separate operations, based on three basic primitives:

- `type α promise`
- `val promise : unit → α promise`
- `val future : α promise → α`
- `val fulfill : α promise × α → unit`

A promise is an explicit handle for a future. Creating one by calling `promise` virtually states the assurance that a suitable value determining that promised future will be made available at some later point in time, fulfilling the promise. The future itself is obtained by applying the `future` function to the promise. A promised future is not replaced automatically, but has to be eliminated by explicitly applying the `fulfill` function to its promise. A promise may only be fulfilled once, any further attempt will raise the exception `Promise`.

Promises allow for partial and top-down construction of data structures with holes, e.g. a tail-recursive formulation of the append function [RLT+04]. However, they are particularly important for concurrent programming: for example, they can be used to implement streams and channels as lists with a promised tail. They also provide an important primitive for programming synchronisation.

For instance, Alice requires no primitive locking mechanisms, they can be fully bootstrapped from promises plus atomic exchange on references, a variant of the fundamental test-and-set operation [Hal85]:
Chapter 6. Alice Through the Looking Glass

(* mutex : unit → (α → β) → (α → β) *)

fun mutex () = let
    val r = ref () (* create lock *)
    fn f ⇒ fn x ⇒ let
        val p = promise ()
        in
            await (exchange (r, future p)); (* take lock *)
            f x
        finally fulfill (p, ()) (* release lock *)
        end
    end
end

FIGURE 6.1. Mutexes for synchronised functions

val exchange : α ref × α → α

As a demonstrating example, Figure 6.1 presents a function implementing mutex locks for synchronising an arbitrary number of argument functions. The following snippet illustrates its use to ensure non-interleaved concurrent output:

val sync = mutex ()
val f = sync (fn x ⇒ (print "x = "; print x; print "\n"))
val g = sync (fn y ⇒ (print y; print "\n"))
spawn f "A"; spawn g "B"

6.2.4 Modules and types

Futures are not restricted to the core language, entire modules can be futures, too: module expressions can be evaluated lazily or concurrently by explicitly prefixing them with the corresponding keywords lazy or spawn. In Section 6.5, we will see that lazy module futures are ubiquitous as a consequence of the lazy linking mechanism for components.

The combination of module futures and dynamic types (Section 6.4) also implies the existence of type futures. They are touched only by the unpack operation (Section 6.4.1) and by pickling (Section 6.4.2). Touching a type generally can trigger arbitrary computations, e.g. by loading a component (Section 6.5).

6.3 HIGHER-ORDER MODULES

For open programming, good language support for modularity is essential. The SML module system is quite advanced, but still limited. Adopting a long line of work [DCH03, Ler95, Lil97, Rus98], Alice extends it with higher-order functors and local modules (for dealing with packages, Section 6.4).

2Alice defines exp₁ finally exp₂ as syntactic sugar for executing a finaliser exp₂ after evaluation of exp₁ regardless of any exceptional termination.
Less standard is the support for nested and abstract signatures: as in Objective Caml, signatures can be put into structures and even be specified abstractly in other signatures. Abstract signatures have received little attention previously, but they are interesting because they enable the definition of polymorphic functors, exemplified by a generic application functor:

\[
\text{functor } \text{Apply (signature } S \text{ signature } T) (F: S \rightarrow T) (X: S) = F X
\]

Polymorphic functors are used in the Alice library to provide certain functionality at the module level (see e.g. Section 6.7.3). More importantly, nested signatures turn structures into a general container for all language entities, which is crucial for the design of the component system (Section 6.5). The presence of abstract signatures renders module type checking undecidable \cite{Lil97}, but this has not turned out to be a problem in practice.

### 6.4 Packages

When a program is able to import and export functionality dynamically, from statically unknown sources, a certain amount of runtime checking is inevitable to ensure the integrity of the program and the runtime system. In particular, it must be ensured that dynamic imports cannot undermine the type system.

*Dynamics* \cite{Myc83, ACPR95} complement static typing with isolated dynamic type checking. They provide a universal type \texttt{dyn} of ‘dynamic values’ that carry runtime type information. Values of every type can be injected, projection is a complex type-case operation that dispatches on the runtime type found in the dynamic value. Dynamics adequately solve the problem of typed open programming by demanding external values to uniformly have type \texttt{dyn}. We see several hurdles that nevertheless prevented the wide-spread adoption of dynamics in practice: (1) their too fine level of granularity, (2) the complexity of the type-case construct, (3) the lack of flexibility with matching types.

We modified the concept of dynamics slightly: dynamics in Alice, called *packages*, contain *modules*. Projection simply matches the runtime package signature against a static one – with full respect for subtyping. Reusing module subtyping has several advantages: (1) it keeps the language simple, (2) it is flexible and sufficiently robust against interface evolution, and (3) it allows the programmer to naturally adopt idioms already known from modular programming. Moreover, packages allow modules to be passed as first-class values, a capability that is sometimes being missed from ML, and becomes increasingly important with open programming. A formal semantics for packages can be found in \cite{Ros05}.

#### 6.4.1 Basics

A package is a value of the abstract type \texttt{package}. Intuitively, it contains a module, along with a dynamic description of its signature. A package is created by injecting a module, expressed by a structure expressions \texttt{strexp} in SML.\footnote{Since Alice supports higher-order modules, \texttt{strexp} includes functor expressions.}
The signature expression $\text{sigexp}$ defines the package signature. The inverse operation is projection, eliminating a package. The module expression

$$\text{unpack} \ \text{exp} : \text{sigexp}$$

takes a package computed by $\text{exp}$ and extracts the contained module, provided that the package signature matches the target signature denoted by $\text{sigexp}$. Statically, the expression has the signature $\text{sigexp}$. If the dynamic check fails, the pre-defined exception $\text{Unpack}$ is raised.

A package can be understood as a statically typed first-class module as proposed by Russo [Rus00, Rus98], wrapped into a conventional dynamic. However, coupling both mechanisms enables $\text{unpack}$ to exploit subtype polymorphism, which is not possible otherwise, due to the lack of subtyping in the ML core language.

### 6.4.2 Pickling

The primary purpose of packages is to type dynamic import and export of high-level language objects. At the core of this functionality lies a service called pickling. Pickling takes a value and produces a transitively closed, platform-independent representation of it, such that an equivalent copy can be constructed in other processes. Since ML is a language with first-class functions, a pickle can naturally include code. Thanks to packages, even entire modules can be pickled.

One obvious application of pickling is persistence, available through two primitives in the library structure $\text{Pickle}$:

\begin{verbatim}
val save : string × package → unit
val load : string → package
\end{verbatim}

The $\text{save}$ operation writes a package to a file of a given name. Any future occurring in the package (including lazy ones) will be touched (Section 6.2.1). If the package contains a local resource, i.e. a value that is private to a process, then the exception $\text{Sited}$ is raised (we return to the issue of resources in Section 6.5.3). The inverse operation $\text{load}$ retrieves a package from a file.

For example, we can write the library structure $\text{Array}$ to disk:

$$\text{Pickle.save ("array.alc", \text{pack} \ \text{Array} : \text{ARRAY})}$$

It can be retrieved again with the inverse sequence of operations:

$$\text{structure} \ \text{Array1 = unpack Pickle.load "array.alc" : ARRAY}$$

Any attempt to unpack it with an incompatible signature will fail with an $\text{Unpack}$ exception. All subsequent accesses to $\text{Array1}$ or members of it are statically type-safe, the only possible point of type failure is the $\text{unpack}$ operation.

Note that the type $\text{Array1.array}$ will be statically incompatible with the original type $\text{Array.array}$, since there is no way to know statically what type identities are found in a package, and all types in the target signature must hence be considered abstract. If compatibility is required, it can be enforced in the usual ML way, namely by putting $\text{sharing constraints}$ on the target signature:
6.4.3 Parametricity, abstraction safety and generativity

By utilising dynamic type sharing it is possible to dynamically test for type equivalences. In other words, evaluation is no longer parametric \([\text{Rey83}]\). For example,

```
functor F (type \(\tau\)) = unpack load file : (val \(\text{it} : \tau\))
```

is a functor that behaves differently depending on what type \(\tau\) it is passed.

Parametricity is important because it induces strong static invariants about polymorphic types, that particularly guarantee abstraction \([\text{Rey83}]\) and enable efficient type erasing compilation. On the other hand, packages enforce the presence of non-parametric behaviour. Alice thus has been designed such that the core language, where polymorphism is ubiquitous, maintains parametricity. Only the module level employs dynamic type information – module evaluation can be type-dependent. This design reduces the costs for dynamic types and provides a clear model for the programmer: only explicit types can affect the dynamic semantics.

The absence of parametricity on the module level still raises the question of how dynamic typing interferes with type abstraction. Can we sneak through an abstraction barrier by dynamically discovering an abstract type’s representation? For instance:

```
signature S = (type \(\tau\); val \(\text{x : } \tau\))
structure M = (type \(\tau\) = int; val \(\text{x = 37}\)) : S
structure N = unpack (pack M : S) : (S where type \(\tau\) = int)
val \(\text{y = N.x + 1}\)
```

Fortunately, the `unpack` operation will fail at runtime, due to a dynamically generative interpretation of type abstraction: with every abstraction operator `:` evaluated, fresh type names are generated dynamically \([\text{Ros03}]\). Abstraction safety is always maintained, even when whole modules cross process boundaries, because type names are globally unique.

Note that when fully dynamic type generativity is too strong to achieve proper type sharing between processes, the implementation of an abstract type can simply be exported as a pre-evaluated component (Section 6.5.4).

## 6.5 COMPONENTS

Software of non-trivial complexity has to be split into functional building blocks that can be created separately and configured dynamically. Such blocks are called components. We distinguish components from modules: while modules provide name spacing, genericity, and encapsulation, components provide physical separation and dynamic composition. Both mechanisms complement each other.

\(^4\)Alice allows abbreviating signatures `sig ... end` with `( ... )`, likewise for structures.
Alice incorporates a powerful notion of component that is a refinement and extension of the component system found in the Oz language [DKSS98], which in turn was partially inspired by Java [GJS96]. It provides all of the following:

- **Separate compilation.** Components are physically separate program units.
- **Lazy dynamic linking.** Loading is performed automatically when needed.
- **Static linking.** Components can be bundled into larger components off-line.
- **Dynamic creation.** Components can be computed and exported at runtime.
- **Type safety.** Components carry type information and linking checks it.
- **Flexibility.** Type checking is tolerant against interface changes.
- **Sandboxing.** Custom component managers enable selective import policies.

### 6.5.1 Introduction

A program consists of a – potentially open – set of components that are created separately and loaded dynamically. Static linking allows both to be performed on a different level of granularity by bundling given components to form larger ones. Every component defines a module – its **export**, and accesses an arbitrary number of modules from other components – its **imports**. Import and export interfaces are fully typed by ML signatures.

Each Alice source file defines a component. Syntactically, it is a sequence of SML declarations that is interpreted as a structure body, forming the export module. The respective export signature is inferred by the compiler. A component can access other components through a prologue of import declarations:

```sml
class spec
end

from string
```

The SML signature specification `spec` in an import declaration describes the entities used from the imported structure, along with their type. Because of Alice’s higher-order modules (Section 6.3), these entities can include functors and even signatures. The string contains the URL under which the component is to be acquired at runtime. The exact interpretation of the URL is up to the component manager (Section 6.5.3), but usually it is either a local file, an HTTP web address, or a virtual URL denoting local library components. For instance:

```sml
import structure Server : sig val run : (α→β) → (α→β) end
from “http://domain.org/server”
```

For convenience, Alice allows the type annotations in import specifications to be dropped. In that case, the imported component must be accessible (in compiled form) during compilation, so that the compiler can insert the respective types.

### 6.5.2 Program execution and dynamic linking

A designated **root** is the main component of a program. To execute a program, its root component is evaluated. Loading of imported components is performed **lazily**, and every component is loaded and evaluated only once. This is achieved
by treating every cross-component reference as a lazy future (Section 6.2.2). The process of loading a component requested as import by another one is referred to as dynamic linking. It involves several steps:

- **Resolution.** The import URL is normalised to a canonical form.
- **Acquisition.** If the component is being requested for the first time, it is loaded.
- **Evaluation.** If the component has been loaded afresh, its body is evaluated.
- **Type Checking.** The component’s export signature is matched against the respective import signature.

Each of the steps can fail: the component might be inaccessible or malformed, evaluation may terminate with an exception, or type checking may discover a mismatch. Under each of these circumstances, the respective future is failed with a standard exception that carries a description of the precise cause of the failure.

### 6.5.3 Component managers and sandboxing

Linking is performed with the help of a **component manager**, which is a module of the runtime library, similar to a class loader in Java [GJS96]. It is responsible for locating and loading components, and keeping a table of loaded components. In an open setting it is important to be able to deal with untrusted components. For example, they should not be given write access to the local file system. Like Java, Alice can execute components in a **sandbox**. Sandboxing relies on two factors: (1) all resources and capabilities a component needs for execution are **sited** and have to be acquired via import through a component manager (in particular, they cannot be stored in a pickle); (2) it is possible to create custom managers and link components through them. A custom manager can never grant more access than it has itself. A custom manager hence represents a proper sandbox.

### 6.5.4 Dynamic creation of components

The external representation of a component is a pickle. It is hence possible to create a component not only statically by compilation, but also dynamically, by a running Alice program. In fact, a pickle created with the `Pickle.save` function (Section 6.4.2) is a component and can be imported as such.

The ability to create components dynamically is particularly important for distribution (Section 6.7.3). Basically, it enables components to capture dynamically obtained information, e.g. configuration data or connections to other processes.

### 6.6 DECOMPOSING COMPONENTS

What are components? The close relation to concepts presented in previous chapters, like modules, packages and futures is obvious, so one might hope that there exists a simple reduction from components to simpler concepts. And indeed, components are merely syntactic sugar. Basically, a component defined by a sequence of declarations `dec` is interpreted as a higher-order procedure:
fn link ⇒ pack struct dec end : sigexp

where link is a reserved identifier and sigexp is the component signature derived by the compiler (the principal signature). In dec, every import declaration

import spec from s

is rewritten as

structure strid = lazy unpack link s : sig spec end
open strid

where strid is a fresh identifier. The expansion makes laziness and dynamic type checking of imports immediately obvious. Component acquisition is encapsulated in the component manager represented by the link procedure. Every component receives that procedure for acquiring its imports and evaluates to a package that contains its own export. The link procedure has type string → package, taking a URL and returning a package representing the export of the respective component. Imports are structure declarations that lazily unpack that package.

When a component is requested for the first time the link procedure loads it, evaluates it and enters it into a table. Figure 6.2 contains a simple model implementation that assumes existence of two auxiliary procedures resolve, for normalising URLs relative to the URL of the parent component, and acquire for loading a component. The parent URL is required as an additional parameter to enable the respective resolution. To achieve proper re-entrancy, the manager uses promises to implement locking on the component table (Section 6.2.3), and unlocks it before evaluating the component (hence the lazy application).

Giving this reduction of components, execution of an Alice program can be thought of as evaluation of the simple application

link "." root

where link is the initial component manager and root is the URL of the program’s root component, resolved relative to the ‘current’ location, which we indicate by a dot here.

6.7 DISTRIBUTION

Distributed programming can be based on only a few high-level primitives that suffice to hide all the embarrassing details of low-level communication.

6.7.1 Proxies

The central concept for distribution in Alice are proxies. A proxy is a mobile wrapper for a stationary function: it can be pickled and transferred to other processes without transferring the wrapped function itself. When a proxy function is applied, the call is automatically forwarded to the original site as a remote invocation, where argument and result are automatically transferred by means of pickling.

5An open declaration merely affects scoping, it does not touch its argument.
val table = ref () : (url × package) list ref

fun link parent url = let
  val url' = resolve (parent, url) (* get absolute URL *)
  val p = promise ()
  val table' = exchange (table, future p) (* lock table *)
in
  case List.find (fn (x,y) ⇒ x = url') table' of
    SOME package ⇒ (fulfill (p, table'); package) (* already loaded *)
    | NONE ⇒ let (* not loaded yet *)
      val component = acquire url' (* load component *)
      val package = lazy component(link url') (* evaluate *)
    in
      fulfill (p, (url',package)::table'); (* unlock *)
      package
  end
end

FIGURE 6.2. The essence of a component manager

Proxies are created using the primitive

val proxy : (α → β) → (α → β)

For instance, the expression proxy (fn x ⇒ x+1) creates a simple proxy.

All invocations of a proxy are synchronous. In order to make an asynchronous call, it suffices to wrap the application using spawn (Section 6.2.1). This immediately returns a future that will be replaced by the result of the remote call automatically once it terminates.

Note that all calls through proxies are statically typed.

6.7.2 Client/server

To initiate a proxy connection, a pickle must be first transferred between processes by other means. The Alice library supports two main scenarios. In the client/server model a client establishes a connection to a known server. A service offered by a server takes the form of a local component, which we refer to as the mobile component. Mobile components can be made available in a network through a simple transfer mechanism adapted from Mozart [Moz04]. To employ it, a component is first packaged (Section 6.4), and then made available for download:

val offer : package → url

Given a package (Section 6.4), this function returns a URL, called a ticket, which publicly identifies the package in the network. A ticket can be communicated to
the outside world by conventional means such as web pages, e-mail, phone, or pigeons. A client can use a ticket to retrieve the package from the server:

```val```
```take : url → package```

The package can then be opened using `unpack`, which dynamically checks that the package signature matches the client’s expectations. Noticeably, this is the only point where a dynamic type check is necessary.

In order to establish a permanent connection, the mobile component must contain proxies. More complex communication patterns can be established by passing proxies back and forth via other proxies. They can even be forwarded to third parties, for instance to enable different clients to communicate directly.

### 6.7.3 Master/slave

In an alternative scenario a master initiates shifting computational tasks to a number of slave computers. The library functor `Run` performs most of the respective procedure: it connects to a remote machine by using a low-level service (such as `ssh`), and starts a slave process that immediately connects to the master. It evaluates a component and sends back the result.

```functor```
```Run (val host : string
signature RESULT
functor F : COMPONENT.MANAGER → RESULT) : RESULT```

`Run` is a polymorphic functor (Section 6.3) with two concrete arguments: the name of the remote host, and a functor that basically defines a dynamic component (Section 6.5.4). It takes a structure representing a component manager, which can be used to access local libraries and resources on the remote host.

We illustrate a two-way connection by sketching the implementation of a distributed computation. A master process delegates computations to slaves that may dynamically request data from the master, by calling the proxy `getData`. For simplicity, we assume that the result of the computation is an integer.

```(* Slaves use getData to acquire specific data. *)
val getData = proxy (fn key ⇒ return associated data)

(* Create a slave process on the given host. *)
fun computeOn (hostname, parameter) = let
  functor Start (CM : COMPONENT.MANAGER) = (val result = computation)
  structure Slave = Run (val host = hostname
    signature RESULT = (val result : int)
    functor F = Start)
  in
  Slave.result
end```

In the definition of `result`, the computation may use `getData`, which is a proxy to the master. It can access local libraries through the component manager `CM`. Note
also that the computation is parameterised by the argument parameter. Then, the following code suffices to perform distributed computations in parallel.

```scala
val res1 = spawn computeOn (“machine1”, parameter1)
val res2 = spawn computeOn (“machine2”, parameter2)
```

The extended version of this paper contains a more extensive example [RLT+04].

### 6.8 IMPLEMENTATION

An implementation of Alice must meet two key requirements: dealing efficiently with the future-based concurrency model, and supporting open programming by providing a truly platform-independent and generic pickling mechanism.

The appropriate technology to achieve platform independence is to use a virtual machine (VM) together with just-in-time (JIT) compilation to native machine code. Futures and light-weight threads are implemented at the core of the system, making concurrency and data flow synchronisation efficient.

Pickling and unpickling are core VM services and available for all data in the store. They preserve cycles and sharing, which is vital for efficient pickling. Alice also features a minimisation mechanism that merges all equivalent subgraphs of a pickled data graph [Tac03].

Code is just heap-allocated data, it is subject to garbage collection and can be pickled and unpickled. The VM allows different internal types of code – e.g. JIT compiled native code and interpreted byte code – to coexist and cooperate. Different codes and interpreters can be selected on a per-procedure basis. Pickler and unpickler automatically convert between these internal codes and a well-defined external format. More details can be found in [RLT+04] and in a technical report [BK02].

Thanks to the pickling-based approach to distribution no complex distributed garbage collection is required. The only inter-process references are proxies. Distributed collection of proxies will be implemented in a future version of Alice.

### 6.9 RELATED WORK

There is numerous work and languages concerned with some of the issues addressed by Alice. A more detailed and extensive comparison can be found in [RLT+04].

Java [GJS96] was the first major language designed for open programming. It is object-oriented and open programming is based on reflection, which allows other components to exploit type information constructively. We feel that general reflection is expensive, invites abuse, and in practice demands a rather limited type system, while packages avoid these issues. Concurrency and serialisation require considerable support from the programmer, code cannot be serialised. No structural type checks are performed when a class is loaded, subsequent method calls may cause a NoSuchMethodError, undermining the type system.
Scala [Ode05] is a hybrid object-oriented/functional language on top of the Java infrastructure. It has a very powerful static type system with expressive abstraction mechanisms. For example, bounded views can specify a lower bound as well as an upper bound for an abstracted type. For comparison, a signature in Alice is a lower bound of the abstracted component. Still, concurrency and distribution are inherited from Java and suffer from the same shortcomings. In particular, the expressiveness of the type system does not carry over to dynamic typing, because Scala types are erased to Java types during compilation.

Oz/Mozart [Smo95, Moz04, VH04] inspired many of the concepts in Alice. Oz has very similar high-level support for concurrency, pickling and components. The Mozart distribution subsystem is more ambitious than Alice, supporting distributed state and futures, for the price of considerably higher complexity. Unlike Alice, Oz is based on a relational core language. It has no type system.

Acute [SLW+04] is probably closest in spirit to Alice. It is an experimental ML-based language for typed open programming that guarantees safe interaction between distributed applications, although the details of distribution are left to the programmer. Pickling allows resources to be dynamically rebound, no respective safety mechanism is built in. Components support versioning, but look more ad-hoc compared to Alice. Unlike in Alice, abstractions can be broken by explicit means, for the sake of flexible versioning.

JoCaml [FMS01] is an innovative distributed extension of OCaml based on the Join calculus. Concurrency and distribution uses channels and is more high-level than in Alice, allowing for complex declarative synchronisation patterns and thread migration. However, JoCaml is not open: pickling is restricted to monomorphic values stored on a global name server and there is no explicit component concept.

CML [Rep99] is a mature concurrent extension of SML. It is based on first-class channels and synchronisation events, where synchronization has to be fully explicit. CML does not address distribution or other aspects of open programming.

Erlang [AVWW96] is a language designed for embedded distributed applications, applied successfully for industrial telecommunication. It is purely functional with an additional, impure process layer. Processes communicate over implicit, process-global message channels. Erlang is untyped, but has a rich repertoire for dealing with failure. It is not designed for open programming and does not directly support code mobility, but a distinctive feature is code update in active processes.

6.10 OUTLOOK

We presented Alice, a functional language for open programming. Alice provides a novel combination of coherent features to provide concurrency, modularity, a flexible component model and high-level support for distribution. Alice is strongly typed, incorporating a module-based variant of dynamics to embrace openness. It is fully implemented with a rich programming environment.
and some small to medium-size demo applications have already been implemented with it, including a distributed constraint solver, a multi-player network game, and a simple framework for web scripting.

There is not yet a formal specification of the full language. Moreover, the implementation does not yet provide extra-lingual safety and security on the level of pickles. To that end, byte code and heap need to carry sufficient type information to allow creation of verifiable pickles. It was a deliberate decision to defer research on these issues to future work.

Acknowledgements. We thank our former colleague Leif Kornstaedt, who co-designed Alice and also invested invaluable amounts of work into making it fly.

REFERENCES


Chapter 7

Experiments with GHC’s Optimiser

László Németh

Abstract: There are several tricky aspects of compiling a functional language: what optimisations to perform, how to do them efficiently, and when, or more precisely in what order, the optimisations should be done. The literature discussing the first question is abundant, for the second is somewhat sparse and for the third is almost non-existent.

This paper uses the Glasgow Haskell Compiler (GHC) to investigate when the optimisations should be performed to gain maximum benefits, where maximum benefits is defined as shortest runtime, for a given set of optimisations. We focus on GHC, because it has been developed as a test bed for compiler research, but we stress that the techniques are applicable to any other compiler for a functional language.

7.1 INTRODUCTION

The optimising part of compilers is an area where theory and practice rarely meet: the first hardly reaches the complexities (non-confluent rewrite rules, undecidable properties as side conditions) involved in compiling a practical language, the second fails to penetrate its depth because the sequence of transformations that take place can only be followed on small chunks of code. On larger examples unexpected interactions take place which manifest itself by a program going much slower (or faster!) and/or allocating a lot more memory after an innocent change to the source or to the compiler or to the runtime system [Wad87].

The difficulty lies in the large number of ways even a few optimisations can be combined. Furthermore, a program transformation may prove to be an optimisation only if other transformations are also used and the order of the two

1Department of Computer Science, Bilgi University, Kurtuluşderesi Caddesi No: 47 Dolapdere 34440 Istanbul, Turkey; E-mail: l nemeth@cs.bilgi.edu.tr
transformation is right. In this paper, we describe a method to explore a large subset of the entire search space: using exhaustive search combined with a handful of constraints we are able to systematically generate all versions of a few programs, examine all the steps of their compilation, their run time and total allocation. From this data we are able to justify conjectures, suggest better sequences, and revisit transformations which have previously been considered uncommon.

7.2 CONTRIBUTIONS OF THE PAPER

Better sequences. We demonstrate the existence of optimisation sequences which not only produce faster code on a range of programs but at the same time compile faster.

Feasibility of the approach. Using cheap hardware and networks, plus some reasonable constraints on the search space we show that the search for better optimisation can be done exhaustively.

Folklore. We compare and contrast the good sequences with previous claims about ordering of the optimisation passes. Some claims turn out to be justified while others appear to be unfounded in the light of our comparisons.

Thorough test of the compiler. With the exhaustive search for better sequences we subjected GHC to a thorough test which exposed bugs both in the specialisation and the common-subexpression passes. Once all the benchmarks compile all possible ways the trust in the correctness of the compiler will increase.

7.3 THE SETUP

7.3.1 Language and implementation

Haskell is a polymorphically typed, lazy, purely functional programming language widely used as a test bed for research. The Glasgow Haskell Compiler (GHC) is a highly optimising ‘industrial-strength’ compiler for Haskell. The compiler itself is written in Haskell and it is widely considered to be the fastest available implementation of a lazy functional language. The speed of its compiled code is due to a range of optimisations and aggressive inlining. GHC performs on a pure, explicitly-typed, intermediate representation of the program called Core.

Some of the Core-to-Core transformations are global, module-at-a-time passes, such as strictness analysis, or let-floating. A handful of other transformations, like case based transformations and beta-reduction, are purely local, and are collected together into a single pass called the simplifier. It is not easy to generate ‘clean’ code, e.g. no dead code, while in the midst of a complicated global pass, so the simplifier is also used to clean up (drop dead code) after global transformations.
The local transformations implemented by the simplifier often cascade: inlining, for example, can expose opportunities for certain case transformations, which in turn can make a call site more attractive for further inlining. The occurrence analysis/simplify pass is therefore iterated until the simplifier indicates that no transformations occur or some arbitrary limit (currently 4) is reached.

In contrast to the vast number of suggested optimisations for lazy functional languages, GHC achieves its performance with only eight global transformations. Two of the transformation passes are concerned with the placement of let bindings: floating outwards (CoreDoFloatOutwards aka full-laziness) increases the scope of let bindings to maximise sharing while floating inwards (CoreDoFloatInwards), almost the exact opposite, decreases the scope in the hope that certain allocations can be avoided. Specialisation (CoreDoSpecialising) eliminates overloading by generating monomorphic versions of polymorphic functions at predescribed types. Strictness analysis (CoreDoStrictness) and the worker-wrapper transformation (CoreDoWorkerWrapper) go hand-in-hand: the first annotates functions with strictness properties, the second exploits the annotation by splitting the body into a (possibly large) worker and a small, therefore inlineable, wrapper. Common-subexpression elimination (CoreCSE) is the same as in imperative languages. Case-liberation (CoreLiberateCase) unrolls recursion to allow evaluations to be lifted from a loop, and finally there is (CoreDoSpecConstr) which specialises recursive functions over constructors so certain arguments of a function can be passed unboxed.

These passes can be run in any order, as many times as it is considered necessary. When instructed to optimise (-O2 on the command-line) the compiler performs a sequence of length 18 of these passes as shown in Table 7.1.

7.4 THE METHOD

We systematically generate all sequences of the passes (7 out of the 8 excluding constructor specialisation) between length 5 and 12. This brute force approach gives rise to an enormous ($\sum_{n=5}^{12} 7^n$) number of possible sequences. Most sequences are fortunately meaningless:

- The global, module-at-a-time passes alter the code depending on a set of conditions which do not change during compilation (some of which can be tweaked on the command-line), and there is no random element – like randomly rearranging the order of two lets – involved. Consequently, after a single run of a given pass, since the side conditions do not change, the second pass would find nothing to do.

- The simplifier’s activities can also be influenced on the command-line, but as we mentioned earlier, a simplifier run means iterate until fixpoint or a predefined limit (4 iterations) is reached. A second run, at the same inlining phase, has nothing to do.
<table>
<thead>
<tr>
<th>CoreDoSimplify Gently 4</th>
<th>one pass (which is allowed to iterate four times) of gentle simplification in which very little inlining [PJM92] takes place. The case-of-case transformation is disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreDoSpecialising</td>
<td></td>
</tr>
<tr>
<td>CoreDoFloatOutwards False False</td>
<td>full-laziness with floating of lambdas and constants are disabled</td>
</tr>
<tr>
<td>CoreDoFloatInwards</td>
<td>moves let’s inwards in the hope of avoiding allocation</td>
</tr>
<tr>
<td>CoreDoSimplify 2 4</td>
<td>simplification with inline phase 2, with a maximum of four iterations</td>
</tr>
<tr>
<td>CoreDoSimplify 1 4</td>
<td>another pass of simplification, with more aggressive inlining</td>
</tr>
<tr>
<td>CoreDoSimplify 0 4</td>
<td>simplification. All identifiers are allowed to be inlined with the exception of those which are specifically marked NoInline</td>
</tr>
<tr>
<td>CoreDoStrictness</td>
<td>annotates identifiers with strictness information</td>
</tr>
<tr>
<td>CoreDoWorkerWrapper</td>
<td>worker-wrapper split</td>
</tr>
<tr>
<td>CoreDoGlomBinds</td>
<td>is not an optimisation on its own right as it lumps together lets to a big letrec</td>
</tr>
<tr>
<td>CoreDoSimplify 0 4</td>
<td>simplification at phase 0</td>
</tr>
<tr>
<td>CoreDoFloatOutwards False True</td>
<td>another pass of full-laziness, this time allowing constants to be floated outwards</td>
</tr>
<tr>
<td>CoreCSE</td>
<td>common-subexpression elimination</td>
</tr>
<tr>
<td>CoreDoFloatInwards</td>
<td></td>
</tr>
<tr>
<td>CoreDoSimplify 0 4</td>
<td></td>
</tr>
<tr>
<td>CoreLiberateCase</td>
<td>case liberation</td>
</tr>
<tr>
<td>CoreDoSpecConstr</td>
<td>constructor specialisation</td>
</tr>
<tr>
<td>CoreDoSimplify 0 4</td>
<td>final simplification</td>
</tr>
</tbody>
</table>

**TABLE 7.1. Current sequence of optimisations for –O2 in GHC**

Imposing the following constraints on the sequences cuts down the number dramatically to 43680 for 7 passes of length 5 – 12.

- No consecutive runs of the same pass: the global transformations and the simplifier do as much as they can in one pass and afterwards there are no more opportunities for the same transformation.

- The worker-wrapper transformation is run immediately after strictness analysis as this ordering gives the most benefit: strictness is an analysis and the worker-wrapper transformation only exploits its results. It is not meaningless to run the worker-wrapper pass without strictness analysis or to allow other transformations to take place as some transformations can ‘manually’ annotate functions with strictness information, but it is very unlikely to be more beneficial.

- Simplifier phases form a non-increasing sequence: counting downwards from phase 2 to phase 0 (inline all identifiers). GHC already follows this rule.

---

\(^2\)For all 8 passes the number of meaningful sequences is 755370, which requires almost 20 times more work, but still manageable.
Chapter 7. Experiments with GHC’s Optimiser

The simplifier is allowed to run a maximum of 10 iterations instead of 4. This ensures that we never have to run two passes of the simplifier consecutively with the same phase.

The limits are arbitrary: any sequence under length 5 is unlikely to be optimising on a sufficiently wide range on programs, and 12 was chosen to control the time it takes to run the experiments.

The generated sequences are used to compile the benchmark programs. The standard libraries are not recompiled.

7.4.1 Benchmark suite

The nofib [Par92] suite of Haskell programs is a large collection of programs which is generally used for benchmarking. The suite is organised into three sets: the imaginary part contains small programs which are rarely true indicators of the optimising abilities of a compiler. The spectral subset has somewhat larger programs, and in some cases can be used to test whether an optimisation is worth the effort. Finally the real subset has programs that were written to get a job done. The programs are usually of reasonable size, and none have trivial input or behaviour.

All the six programs of Table 7.2 come from the real part of the nofib suite and were chosen to allow a rough comparison – by reporting speedups for the very same programs – of our method, which can be seen as a relatively high level optimisation, to that of Mycroft [NM02] who examined how much speed improvement can be gained by paying attention to the cache behaviour of lazy functional programs. The two approaches otherwise are complementary.

While the programs are unmodified, their inputs were chosen so that runtimes are not too short.

We believe the question whether the chosen benchmarks are representative (of programs that get written in Haskell?) to be somewhat misguided since it suggests the existence of a sequence of optimisations which can compile any code ‘optimally’, say to run not slower than 5% of the runtime of the best sequence for a given set of transformations. While it might be possible to construct such a simplifier, for example by making all known exceptions a special case with special side conditions, it is not very desirable: the code which implements it would likely to be horrendously complicated and unmaintainable and it would not deepen our understanding of what constitutes a good sequence.

Instead, we suggest that the compiler should adapt its sequence of optimisations to the code being compiled based on some syntactic or semantic criteria. Consider, for example, the difference between compiling library code versus the main module of an application with respect to specialisation. At the time of compiling the code which is destined to be a library it is rarely known what monomorphic instances will be required. So instead of not specialising at all or generating all the frequently used instances the code should be compiled in such a way that it is prepared to be specialised at a later stage without missing possibilities of optimisation. We do not pursue these ideas any further, but it appears that the brute
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Lines</th>
<th>Runtime –O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>Frontier-based strictness analyser</td>
<td>5740</td>
<td>26.30s</td>
</tr>
<tr>
<td>compress2</td>
<td>Text compression</td>
<td>147</td>
<td>0.36s</td>
</tr>
<tr>
<td>fulsom</td>
<td>Solid modeller</td>
<td>857</td>
<td>1.02s</td>
</tr>
<tr>
<td>hidden</td>
<td>PostScript polygon renderer</td>
<td>362</td>
<td>1.62s</td>
</tr>
<tr>
<td>infer</td>
<td>Hindley-Milner type inference</td>
<td>561</td>
<td>10.40s</td>
</tr>
<tr>
<td>scs</td>
<td>Set circuit simulator</td>
<td>585</td>
<td>3.10s</td>
</tr>
</tbody>
</table>

TABLE 7.2. Benchmarks used

The brute force approach of this paper is an appropriate tool in finding distinct classes of programs and their optimisation sequences.

### 7.4.2 Compilation

The benchmarks were compiled, using the same modified version of GHC 6.2.2, on a network of practically identical machines: Pentium IV 2.66GHz, 512KB cache, 512MB RAM, connected through 100MBps Ethernet, running a recent version of Debian Linux with kernel 2.4.23.

Using the optimisation options of the fastest contenders on each individual machine programs were recompiled on a machine not connected to any type of network and rerun (3 times, and the average is taken) to ensure comparable results.

### 7.5 EVALUATION

Feasibility of the brute force approach hinges on four issues: (1) space and time requirements of compiling the programs all the required ways; (2) the time it takes to select the best performers on each machine plus the time it takes to recompile and rerun the candidates on a single machine for comparable runtimes; (3) the existence of ‘better’ sequences; and finally (4) the applicability of these good sequences to other programs. We examine these four issues in turn.

#### 7.5.1 Space and time requirement

Compiling the six benchmark programs 43680 different ways takes about 15 hours on 50 computers, which is not unreasonable. The size of the log files of the compilation, which include the sequence of the passes, statistics about the simplifier, and the final simplified version of each module in Core (which is not necessary for the purposes of this paper) is large (1MB+) but compresses extremely well with the Burrows-Wheeler block sorting algorithm of bzip2. The size of the compressed

---

3 Being ‘practically identical’ is not very important as long as the machines are the same architecture, because the fastest, selected versions are recompiled and rerun on the same machine. The machines were of identical hardware specification, supplied by one manufacturer around the same date, and had identical software installed. See Section [AS2].
logs varies between 20-150KB. Stripped executables, also compressed, are about 100K. Log files of the actual runs, which include basic information about total allocation and the output of the `time` command are tiny, approximately 350 bytes. This puts the overall space requirement to around 80GB for the six benchmarks.

Neither the space nor the time requirements are excessive. Adding a dozen more benchmarks or a few more optimisation passes is still manageable. Space can be saved by not logging the final simplified version of each module and/or deleting the executables if compilation is interleaved with running the benchmarks.

### 7.5.2 Recompilation and benchmarking

While our method is essentially an exhaustive search the results (especially run times) from different machines are not directly comparable. To ensure valid comparison we choose (the sequences of) the top performers from each machine and then recompile the programs on one machine.

There are various ways the best candidates on each machine can be chosen. Whatever the choice is the main limitation is that the chosen programs should compile and run on a single machine in reasonable time. We found that on average a Pentium IV can compile about 250 programs (of our chosen benchmarks) per hour.

1. We can take the top $n$ performers with a fixed $n$. For $m$ machines and $b$ benchmarks this gives $n \times m \times b$ compilations. Recompiling the top 20 performers is a day’s work.

2. We can compile and run the six benchmark programs on each machine with GHC’s current optimisation sequence (\(-O2\)) and choose all those programs which are faster. Approximately, 5% of the generated sequences (2000 programs out of 43680) make the benchmarks go faster than \(-O2\). Recompilation is about two day’s work.

We currently use the first option as the second one is generally a lot more work unless there are few sequences better than \(-O2\).

The time to rerun all the benchmarks is considerably less than the time it takes to compile them, even when the inputs are suitably modified.

### 7.5.3 Good sequences

Table 7.3 lists the actual speedups for all six programs for the best sequence of the given benchmark. As expected, there is no sequence which makes all the programs fastest and there is only one (see Section 7.5.4) which puts all six benchmarks into the top ten. Approximately 1% of the sequences make all benchmarks perform in the top one hundred, all of which beats \(-O2\). Unfortunately, the difference of the averages across the benchmarks is so little – well below for example the accuracy of the `time` command we use to measure the runtime – that it is
impossible to identify a single sequence as the best. The table shows modest, but not negligible improvements. As the slowest performers are much slower than \(-O2\) it also shows that GHC’s sequence is impressively tuned. Table 7.3 is not without surprises: \texttt{anna} for example when compared to unoptimised code gets slower after optimisation. Fortunately this happens with only one sequence.

Sequences in Table 7.4 which gives a random identifier, the length of the sequence, the sequence itself and a short remark, come from the top 100 performers, and all of them are faster than \(-O2\). They were randomly chosen to demonstrate that widely differing sequences are amongst the best. There are number of points worth noting about the entries:

- Full-laziness (FloatOutwards) generally comes late, despite the observation of Santos [San95] and the current practice in GHC (Table 7.1) that it should happen before any inlining takes place.

- CSE features rather frequently (253 out of 292 examined) despite Chitil’s claim [Chi97] that common-subexpressions are uncommon in a lazy functional programs. Because of the exhaustiveness of the search (for all sequences with CSE one without it was run and did not make it to the top) we can make a strong claim: CSE does make (a yet unquantified) difference.

- The number of simplifier passes varies from as little as three to six, not counting the initial ‘Gentle’ pass.

- One of the best performers is only of length 11, which compares well with GHC’s current sequence of length 18.

- Strictness analysis and the worker-wrapper transformation appears in all the top performers, though its placement varies considerably.

A more refined analysis of groups of ‘good’ and ‘bad’ sequences have been attempted through searching for longest common subsequences and other similarity measures over sequences but none of them proved to be conclusive so far. A detailed study of good and bad sequences for space behaviour (total allocation, lag, drag, and void) is the subject of a forthcoming paper.
<table>
<thead>
<tr>
<th>ID</th>
<th>(length) Options</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>nipwxc</td>
<td>(11) Simplify, FloatInwards, Simplify 2, FloatOutwards, Specialising, Simplify 1, Strictness, WorkerWrapper, LiberateCase, Simplify 0, CSE</td>
<td>short sequence, late CSE</td>
</tr>
<tr>
<td>rYfleO</td>
<td>(14) Simplify, Specialising, Simplify 2, LiberateCase, Simplify 1, CSE, Simplify 0, FloatInwards, Simplify 0, FloatOutwards, Simplify 0, Strictness, WorkerWrapper, CoreDoSimplify 0</td>
<td>full-laziness after inlining</td>
</tr>
<tr>
<td>kgbeGv</td>
<td>(11) Simplify, Specialising, Strictness, WorkerWrapper, Simplify 2, LiberateCase, Simplify 1, CSE, FloatInwards, Simplify 0, FloatOutwards</td>
<td>late full-laziness</td>
</tr>
<tr>
<td>TMM95z</td>
<td>(11) Simplify, LiberateCase, FloatInwards, Simplify 2, FloatOutwards, Simplify 1, CSE, Simplify 0, Strictness, WorkerWrapper, Simplify 0</td>
<td>no specialisation</td>
</tr>
<tr>
<td>r4qQTC</td>
<td>(13) Simplify, CSE, Simplify 2, LiberateCase, Simplify 1, Specialising, Simplify 0, FloatOutwards, Simplify 0, Strictness, WorkerWrapper, FloatInwards, Simplify 0</td>
<td>early CSE</td>
</tr>
<tr>
<td>NtmAht</td>
<td>(12) Simplify, Specialising, Simplify 2, Strictness, WorkerWrapper, Simplify 1, LiberateCase, Simplify 0, FloatInwards, FloatOutwards, Simplify 0, CoreCSE</td>
<td>floating in is not followed by simplification</td>
</tr>
<tr>
<td>zPYd7H</td>
<td>(14) Simplify, Specialising, Simplify 2, CSE, Simplify 1, LiberateCase, Simplify 0, FloatInwards, Simplify 0, FloatOutwards, Simplify 0, Strictness, WorkerWrapper, CoreDoSimplify 0</td>
<td>floating in before strictness</td>
</tr>
<tr>
<td>HP0d25</td>
<td>(12) Simplify, FloatOutwards, FloatInwards, Simplify 2, Specialising, Simplify 1, CSE, Simplify 0, LiberateCase, Strictness, WorkerWrapper, Simplify 0</td>
<td>the only early full-laziness</td>
</tr>
<tr>
<td>N0vZ5h</td>
<td>(13) Simplify, Strictness, WorkerWrapper, Simplify 2, CSE, Simplify 1, LiberateCase, Simplify 0, Specialising, Simplify 0, FloatOutwards, FloatInwards, Simplify 0</td>
<td>very early strictness</td>
</tr>
<tr>
<td>tYHSxT</td>
<td>(11) Simplify, Specialising, FloatInwards, LiberateCase, Simplify 2, CSE, FloatOutwards, Simplify 1, Strictness, WorkerWrapper, Simplify 0</td>
<td>late strictness</td>
</tr>
</tbody>
</table>

**TABLE 7.4. Ten sequences better than \(-O2\)**

### 7.5.4 How general are they?

So far comparisons have been limited to other programs from the `nofib` suite. While this is not ideal it does give an indication about the generality of the sequences which were derived from only six programs. Table 7.5 summarises our findings and they can be interpreted as follows. In the first column the name of the benchmark program appears. For each program five rows of data are presented: the first row (\(-O2\)) gives details about compiling the given benchmark with \(-O2\). The second row gives the name of two sequences which minimise/maximise the size of the generated code (the third column). The third row shows min/max values of compilation time (the fourth column), the fourth shows min/max values of total allocation and so on. As these extremal values do not necessarily coincide for a given sequence (there is no guarantee that the compiled code size and the compile time and the total allocation and the runtime are minimum or maximum for a given sequence) the minimum/maximum rela-
tionship between the numbers of an entry only holds in the diagonal where the items are highlighted: in other columns the first/second number shows the allocation, run time etc. of the sequence. For example, the fourth entry for circsim in Table 7.5 which starts HP0d25/kgbeGv says that the sequence HP0d25 makes circsim allocate the least amount (612358260) bytes of memory while the other sequence kgbeGv makes it allocate the most (641035716) for the sequences of Table 7.4. Note, that the sequences in Table 7.4 were chosen to demonstrate that widely differing options generate fast code and not to show the very best performers in each category. The meaning of columns is self explanatory: the first (Sequence) gives the name of the sequence from Table 7.4 which was used to compile the program. Consequent columns can be grouped to (1) compile time information giving the size of the code (denoted Size) and compilation time (Compile), (2) runtime information giving the total allocation (Alloc) and the total runtime of the program (Run).

Within the six programs of Table 7.3 the length 11 sequence ‘Simplify, FloatInwards, Simplify 2, Strictness, WorkerWrapper, Simplify 1, LiberateCase, Simplify 0, CSE, FloatOutwards, Simplify 0’ (which does not appear in Table 7.4) makes all six programs perform in the top ten. While the existence of such sequence is encouraging, there is no guarantee that the same sequence will not make another program run far worse than with -O2.

Comparing the good sequences to other programs gives somewhat mixed results. Table 7.5 and 7.6 shows twenty-two programs 4 from all three parts (imaginary, spectral, and real) of the nofib suite. For most programs the better sequences found only examining six programs do give improvements. The top sequences still perform well for cacheprof and symalg (both from the real subset), though the speedups are not as great as in Table 7.3. On the other hand none of the top sequences make circsim run as fast as with -O2 and in general the gap between the best of the top ten and -O2 is much narrower (<2%). This can be attributed to several different causes: either the sequences derived for the six real benchmarks are not general and possibly do not generalise well, or a larger set of programs must be used to find better sequences or GHC’s optimisation sequence is fine tuned to the spectral subset of the nofib suite. All three explanations are plausible and warrant further investigation.

There are real surprises as well: the sequence HP0d25 makes constraints run 25% faster despite that at the same time it also makes it allocate most. It is unclear whether -O2 misses an opportunity or a bug in the compiler perhaps triggers possibilities of further optimisation.

7.6 RELATED WORK

There have been a number of attempts at meta optimisation, optimising the optimiser of compilers. The first thorough study for lazy functional languages on the

---

4Results for the entire nofib and detailed analysis of optimisation on the code of small programs can be found in the extended version of this paper.
effect of individual transformations on the quality of the code was conducted by Santos and reported in his thesis [San95]. His results were derived by studiously examining large chunks of core programs and fine tuning both the transformations, that is the side conditions on the rewrite rules, and their sequence. His method is inherently limited as optimisations interact unexpectedly.

Realising that ‘eyeballing’ large amounts of optimised code is not feasible, for imperative languages Pugh’s Unified Transformation Framework (UTF) [KP93] allows searching the entire optimisation space. Based on UTF, machine learning techniques have been employed to effectively search for better sequences [LO04].

Closely related is iterative compilation [KKO02], where the compiler repeatedly executes different versions of the code and uses the feedback to decide the next optimisation attempt.

All these approaches suffer from the same limitation: none can prove definitively that for a given set of optimisations the one found is the best for a class of programs or a given program.

7.7 CONCLUSIONS

This paper has revisited an ancient approach to meta optimisation, previously seen as computationally infeasible. With the addition of sensible constraints on the optimisation space and the increased availability of cheap hardware and networks, the method has been demonstrated to be viable.

We not only found better sequences of optimisation passes which make programs compile and run faster at the same time, almost as a side effect we subjected GHC to the most thorough test of thousands of compilations with different optimisations. Several bugs in the specialisation and common-subexpression elimination pass showed up. Once eliminated, the absence of these bugs will increase our trust in the compiler.

Future work is needed in several directions. The main bottleneck of our approach is the recompilation and rerunning of the benchmarks on a single machine to ensure comparable results. Thorough statistical approaches it appears to be possible to extend benchmarking on a single machine to a number of machines while ensuring that run times remain comparable. Doing so will further expand our ability to search for longer and even better sequences of optimisations.

Analysis of the results should be put on a more solid theoretical basis to see if the measured differences are statistically significant and due to the optimisation sequence or other factors.

Our main concern in this experiment was to minimise run time of the optimised programs. The analysis should be repeated for space consumption and it would be interesting to explore whether our findings can be correlated to Sands’ improvement theory [MS99] [GS01].

Most importantly perhaps, there is very little understanding why the newly discovered optimisation sequences work better than the one currently used by GHC. Perhaps this thorough study will open up new avenues for understanding optimisation and optimising compilers.
Acknowledgements. Many thanks to Simon Marlow who prompted me to generate Table 7.5 and 7.6 and in doing so made me write my most elaborate awk scripts to date, and to Chris Stephenson for a thorough review of the paper.

REFERENCES


<table>
<thead>
<tr>
<th>Program</th>
<th>Sequence</th>
<th>Size</th>
<th>Compile</th>
<th>Alloc</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O2</td>
<td>242889</td>
<td>1.02</td>
<td>62860</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>ansi</td>
<td>241833/243689</td>
<td>0.83/0.99</td>
<td>62892/62992</td>
<td>0.03/0.01</td>
<td></td>
</tr>
<tr>
<td>TMM95z/Hipod25</td>
<td>243689/243545</td>
<td>0.99/0.94</td>
<td>62892/63053</td>
<td>0.01/0.01</td>
<td></td>
</tr>
<tr>
<td>TMM95z/r4qQTC</td>
<td>243689/242313</td>
<td>0.99/0.90</td>
<td>62892/62932</td>
<td>0.01/0.03</td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>350644</td>
<td>1.11</td>
<td>205663808</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td>atom</td>
<td>348580/350676</td>
<td>0.90/1.11</td>
<td>229581708/205663808</td>
<td>0.58/0.51</td>
<td></td>
</tr>
<tr>
<td>N0vZ5h/TMM95z</td>
<td>348820/348580</td>
<td>0.95/0.90</td>
<td>205663808/229581708</td>
<td>0.52/0.57</td>
<td></td>
</tr>
<tr>
<td>NtmAht/TMM95z</td>
<td>350900/348580</td>
<td>1.09/0.90</td>
<td>205663808/229581708</td>
<td>0.49/0.57</td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>196816</td>
<td>1.29</td>
<td>128972</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>awards</td>
<td>197232/192344</td>
<td>1.28/1.33</td>
<td>129576/129576</td>
<td>0.02/0.02</td>
<td></td>
</tr>
<tr>
<td>banner</td>
<td>254106</td>
<td>1.57</td>
<td>122296</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>213520/232272</td>
<td>2.98/3.83</td>
<td>27398792/27398792</td>
<td>0.09/0.09</td>
<td></td>
</tr>
<tr>
<td>boyer</td>
<td>213056/232272</td>
<td>2.86/3.83</td>
<td>27398792/27398792</td>
<td>0.09/0.09</td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>225552/226640</td>
<td>4.09/3.83</td>
<td>27398792/27398792</td>
<td>0.09/0.09</td>
<td></td>
</tr>
<tr>
<td>constraints</td>
<td>319782/330310</td>
<td>4.23/4.88</td>
<td>634884716/626695220</td>
<td>2.42/2.34</td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>289510</td>
<td>2.34</td>
<td>947010684</td>
<td>7.49</td>
<td></td>
</tr>
<tr>
<td>crypta1</td>
<td>287866/289654</td>
<td>2.11/2.20</td>
<td>956680152/966636052</td>
<td>7.45/7.60</td>
<td></td>
</tr>
<tr>
<td>wave4main</td>
<td>202300</td>
<td>2.47</td>
<td>176013160</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td>293296/314176</td>
<td>9.55/10.80</td>
<td>315968204/315018004</td>
<td>0.77</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 7.5.** Other programs with the best sequences (imaginary, spectral)
<table>
<thead>
<tr>
<th>Program</th>
<th>Sequence</th>
<th>Size</th>
<th>Compile</th>
<th>Alloc</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>208848/217664</td>
<td>3.19/3.62</td>
<td>155107528/154686232</td>
<td>0.64/0.65</td>
</tr>
<tr>
<td></td>
<td>typecheck</td>
<td>208928/217664</td>
<td>3.14/3.62</td>
<td>154698364/155107528</td>
<td>0.64/0.65</td>
</tr>
<tr>
<td></td>
<td>rYfeO/N0vZ5h</td>
<td>209216/213200</td>
<td>3.26/3.46</td>
<td>154043644/169173192</td>
<td>0.66/0.72</td>
</tr>
<tr>
<td></td>
<td>zPYd7H/kgbeGv</td>
<td>212400/210208</td>
<td>3.46/3.18</td>
<td>154047868/169169096</td>
<td>0.62/0.74</td>
</tr>
<tr>
<td>-O2</td>
<td>N0vZ5h/NtmAht</td>
<td>531525/55379791</td>
<td>11.84/13.71</td>
<td>289000342/273080460</td>
<td>1.68/1.62</td>
</tr>
<tr>
<td></td>
<td>simple</td>
<td>531989/553797</td>
<td>11.17/13.71</td>
<td>287004124/273080460</td>
<td>1.75/1.62</td>
</tr>
<tr>
<td></td>
<td>kgbeGv/NtmAht</td>
<td>541235/531525</td>
<td>12.67/11.84</td>
<td>256333652/289000342</td>
<td>1.63/1.68</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/N0vZ5h</td>
<td>549125/531989</td>
<td>13.25/11.17</td>
<td>270302344/287004124</td>
<td>1.56/1.76</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>977074/1036674</td>
<td>57.84/61.68</td>
<td>321941404/319535068</td>
<td>27.09/27.01</td>
</tr>
<tr>
<td></td>
<td>anna</td>
<td>977074/1031026</td>
<td>57.84/120.15</td>
<td>321941404/19408952</td>
<td>27.09/26.94</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/TMM95z</td>
<td>1010066/977074</td>
<td>60.34/57.84</td>
<td>315361796/321491404</td>
<td>26.82/27.27</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/rYfeO</td>
<td>1010066/1002354</td>
<td>60.34/61.73</td>
<td>315361796/318167136</td>
<td>26.82/27.42</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>607603/821623</td>
<td>23.24/34.39</td>
<td>231340688/-</td>
<td>1.02/-</td>
</tr>
<tr>
<td></td>
<td>cacheprof</td>
<td>607603/217664</td>
<td>23.24/34.39</td>
<td>231340688/-</td>
<td>1.02/-</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/N0vZ5h</td>
<td>610935/630279</td>
<td>25.59/24.99</td>
<td>223885704/238472532</td>
<td>1.02/1.03</td>
</tr>
<tr>
<td></td>
<td>r4qQTc/zPYd7H</td>
<td>627543/627447</td>
<td>25.23/25.73</td>
<td>231828152/231234160</td>
<td>0.99/1.05</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>504500</td>
<td>15.59</td>
<td>186944804</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>fulsom</td>
<td>493588/542180</td>
<td>14.07/16.73</td>
<td>236125632/205597488</td>
<td>1.06/0.89</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/N0vZ5h</td>
<td>493588/540626</td>
<td>14.07/17.14</td>
<td>236125632/197304524</td>
<td>1.06/0.89</td>
</tr>
<tr>
<td></td>
<td>r4qQTc/kgbeGv</td>
<td>509268/493588</td>
<td>15.58/14.07</td>
<td>188916744/236125632</td>
<td>0.87/1.06</td>
</tr>
<tr>
<td></td>
<td>rYfeO/kgbeGv</td>
<td>500443/493588</td>
<td>15.37/14.07</td>
<td>193704524/236125632</td>
<td>0.87/1.08</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>501159</td>
<td>10.95</td>
<td>554238104</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>hidden</td>
<td>495831/503159</td>
<td>10.05/10.52</td>
<td>528499288/539923976</td>
<td>1.51/1.53</td>
</tr>
<tr>
<td></td>
<td>N0vZ5h/rYfeO</td>
<td>496135/501047</td>
<td>9.92/10.66</td>
<td>543306832/527243308</td>
<td>1.54/1.54</td>
</tr>
<tr>
<td></td>
<td>rYfeO/HPo0d25</td>
<td>501047/496455</td>
<td>10.66/10.12</td>
<td>527243308/554033936</td>
<td>1.54/1.64</td>
</tr>
<tr>
<td></td>
<td>TMM95z/HPo0d25</td>
<td>495135/496455</td>
<td>10.05/10.12</td>
<td>528499288/554033936</td>
<td>1.49/1.65</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/nipwxc</td>
<td>402662/405958</td>
<td>10.39/10.30</td>
<td>851656892/861338180</td>
<td>10.25/10.71</td>
</tr>
<tr>
<td></td>
<td>infer</td>
<td>402662/405558</td>
<td>10.39/10.76</td>
<td>851656892/862911748</td>
<td>10.25/10.31</td>
</tr>
<tr>
<td></td>
<td>nipwxc/N0vZ5h</td>
<td>403333/404462</td>
<td>10.69/10.43</td>
<td>849057200/863387040</td>
<td>10.02/9.91</td>
</tr>
<tr>
<td></td>
<td>rYfeO/kgbeGv</td>
<td>404662/405558</td>
<td>10.43/10.33</td>
<td>863387040/850928032</td>
<td>9.91/10.73</td>
</tr>
<tr>
<td>-O2</td>
<td>N0vZ5h/rYfeO</td>
<td>556164/562566</td>
<td>12.57/13.89</td>
<td>864031608/864277888</td>
<td>2.92/2.94</td>
</tr>
<tr>
<td></td>
<td>svs</td>
<td>556164/562566</td>
<td>12.57/13.89</td>
<td>864031608/864277888</td>
<td>2.92/2.94</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/r4qQTc</td>
<td>565170/565234</td>
<td>13.62/13.59</td>
<td>860727196/864647380</td>
<td>2.95/3.08</td>
</tr>
<tr>
<td></td>
<td>rYfeO/r4qQTc</td>
<td>565266/565234</td>
<td>13.89/13.59</td>
<td>864277888/864647380</td>
<td>2.84/3.08</td>
</tr>
<tr>
<td>-O2</td>
<td>zPYd7H/NtmAht</td>
<td>463175/473735</td>
<td>10.67/10.83</td>
<td>501383386/501387994</td>
<td>30.78/30.97</td>
</tr>
<tr>
<td></td>
<td>symalg</td>
<td>452502/473303</td>
<td>9.97/10.98</td>
<td>501383352/501383352</td>
<td>30.81/30.96</td>
</tr>
<tr>
<td></td>
<td>HPo0d25/r4qQTc</td>
<td>453655/463767</td>
<td>10.29/10.46</td>
<td>501383349/5013843700</td>
<td>31.00/31.04</td>
</tr>
<tr>
<td></td>
<td>zPYd7H/kgbeGv</td>
<td>463175/465463</td>
<td>10.67/10.37</td>
<td>501383386/5013843564</td>
<td>30.73/31.26</td>
</tr>
<tr>
<td>-O2</td>
<td>TMM95z/NtmAht</td>
<td>925128/1038410</td>
<td>54.73/60.06</td>
<td>523956/523228</td>
<td>0.01/0.01</td>
</tr>
<tr>
<td></td>
<td>veritas</td>
<td>925128/1038410</td>
<td>54.73/60.06</td>
<td>523956/523228</td>
<td>0.01/0.01</td>
</tr>
</tbody>
</table>

**TABLE 7.6.** Other programs with the best sequences (spectral, real)
Abstract: Forms are parts of a graphical user interface (GUI) that show a (structured) value and allow the user to update it. Some forms express a choice between two or more (structured) values using radio buttons or check boxes. We show that explicitly modelling such a choice leads to a cleaner separation of logic and layout. This is done by extending the combinator library FunctionalForms with disjoint form combinators. To implement these, we have generalized the technique of compositional functional references which underlies the library.

8.1 INTRODUCTION

Forms are parts of a graphical user interface (GUI) that show a (structured) value and allow the user to update it. For example, the omnipresent dialogs labelled Options, Settings and Preferences are forms. An address book can also be considered a form. In our previous work, we have developed the combinator library FunctionalForms [EAK05] for building forms in a concise, compositional way.

Many real-life forms allow a choice between two or more alternatives, some of which require extra information. For example, the form in Figure 8.1 indicates whether the user wishes to receive a certain newsletter; if they do, the text entry field next to this option should contain his/her e-mail address. If they do not, this text field is irrelevant (some GUIs provide a visual clue for this: the control is dimmed).

Usually, the information in such a form is processed as a product-like data structure containing the choice (e.g. as a boolean) and the extra information (e.g. as a string). However, most functional languages allow data types which are more
suited to this task, namely disjoint union types. In Haskell [Pey03], we would define

\[
\text{data NewsLetter = NewsYes String | NewsNo}
\]

for the type of information in the example form.

While the combinators in FunctionalForms previously only supported forms with product-like data structures, in this paper we extend them to enable the explicit definition of such a disjoint form. Rather than as a ‘yes/no’ form and an ‘e-mail’ form, it can now be composed as a ‘yes, e-mail’ form or a ‘no’ form. We demonstrate that this technique leads to a better separation of logic and layout in disjoint forms. For its implementation, we have generalized the compositional functional references which underlie the library.

This paper is organized as follows: it first gives a summary of the library’s basic use, which has not changed (Section 8.2). Then, the use and merits of the extension are demonstrated (Section 8.3), after which its implementation is discussed in Section 8.4. Next, we show that the gained flexibility leads to some safety issues (Section 8.5). We finish with related work (Section 8.6) and conclusions (Section 8.7).

### 8.2 FUNCTIONALFORMS SUMMARY

FunctionalForms [EAK05] is a combinator library built on the GUI library wx-Haskell [Lei04] (itself built on the cross-platform C++ library wxWidgets [wxW]). It can be seen as an embedded domain-specific language for forms: it consists of atomic forms and ways to combine them into larger forms in a declarative style. In this section, we give a brief summary of its basic use, which is the same as described in [EAK05], although the types have changed a little.

A form is a GUI part, residing somewhere within a dialog with OK and Cancel buttons, which is only able to display and alter a certain value. When the dialog appears, the form has an initial value which is provided by its environment; subsequently, the user can read and alter this value; at the end, the user closes the dialog with one of the buttons, and the form passes the final value back to the environment. The type of this value is called the subject type of the form.

Atomic forms correspond to a single control containing an editable value. Examples are a text entry field, containing a String, and a spin control, containing an Int:
Chapter 8. Disjoint Forms in Graphical User Interfaces

We follow the convention that library functions are underlined, and an atomic form is named after the corresponding *wxHaskell* function which creates its control, but with an additional prime symbol. Every atomic form is parameterized with a list of optional properties used for customizing the control, e.g. its size and font (leaving this list empty produces reasonable defaults). Some atomic forms, like `spinCtrl'`, require additional parameters: its first two arguments indicate a minimum and maximum value.

All these parameters actually have little to do with *FunctionalForms*: they are directly passed to the *wxHaskell* control creation function. In contrast, the last parameter of both forms is specific to *FunctionalForms*: it is a reference value which relates the atomic form’s subject type (*String* and *Int*, respectively) to the subject type of the top-level form. A more detailed description of the *Ref* and *FForm* types is postponed to Section 8.4.

To combine atomic forms into larger forms, two aspects have to be composed: layout and subject type. The former is performed by layout combinators like `grid'`, `margin'` and `floatLeft'`. These are based on the *wxHaskell* layout combinators after which they are named, but operate directly on forms. For example, the two atomic forms can be put in a grid layout with some labels (see Figure 8.2):

```
grid' 5 5 [ [label "name :", entry' [] name ] , [label "nr. of tickets :", spinCtrl' 1 6 [] nr ] ]
```

Note that the two reference values (*name :: *Ref m String*) and (*nr :: *Ref m Int*) are free variables in this expression. Also, this form’s subject type is not yet established. To complete the form composition, *name* and *nr* are bound in a lambda expression, onto which a subject type combinator, namely `declare2`, is applied:

```
ticketsForm :: Monad m \rightarrow Ref m (String,Int) \rightarrow FForm win m Layout
ticketsForm = declare2 $ \lambda(name,nr) \rightarrow
grid' 5 5 [ [label "name :", entry' [] name ] , [label "nr. of tickets :", spinCtrl' 1 6 [] nr ] ]
```

1 instead of on *Layout* values of widgets – for those familiar with *wxHaskell*
This ‘declares’ \( \text{ticketsForm} \)’s subject type to be \((\text{String}, \text{Int})\), as witnessed by its actual type declaration (which can be omitted). Just like the atomic forms, \( \text{ticketsForm} \) can now be used as a component of a larger form. Note how this two-stage process of form construction separates the definition of layout and subject type structures, providing a great deal of freedom to the library user (see also \[\text{EAK05}\]).

Besides \textit{declare2}, which declares a pair, the library also provides subject type combinators for tuples of higher arity and for lists.

\[
declare2 :: \text{Monad } m \Rightarrow ((\text{Ref } m \ t_1, \text{Ref } m \ t_2) \to z) \to \text{Ref } m (t_1, t_2) \to z \\
declare3 :: \text{Monad } m \Rightarrow ((\text{Ref } m \ t_1, \text{Ref } m \ t_2, \text{Ref } m \ t_3) \to z) \to \text{Ref } m (t_1, t_2, t_3) \to z \\
\ldots \\
declareL :: \text{Monad } m \Rightarrow ([\text{Ref } m \ t] \to z) \to \text{Ref } m [t] \to z
\]

The \textit{declareL} combinator only composes forms for the list \textit{elements} and cannot alter the spine; it produces a form for lists of a fixed length.

To run a form in a \textit{wxHaskell} program, the library function \textit{runInDialog} is used. For example, this runs the above defined \textit{ticketsForm} with \textit{John} and \textit{2} in the atomic forms:

\[
\text{do} \ldots \\
(\text{newname}, \text{newnr}) \leftarrow \text{runInDialog parentWindow ticketsForm ("John", 2)} \\
\ldots
\]

The function takes as its arguments a pointer to a parent window, the form itself, and an initial value of the form’s subject type. It returns an IO action, which produces a modal dialog containing the form and \textit{OK/Cancel} buttons. When the user presses \textit{OK}, the return value is bound to altered value in the form; if \textit{Cancel} is pressed, the initial value is returned instead. After this, the IO thread continues.

### 8.3 Combinators for Disjoint Forms

This section describes, from a library user’s point of view, the additions for defining disjoint forms. As an example, we will define a form for contact information, depicted in Figure 8.3. It has subject type
data Contact = ByPhone Phone | ByEmail String | NotAtAll

and expresses a choice between a phone number, an email address and no information at all. Before we can start defining the form itself, we need to define three custom subject type combinators for this type’s data constructors. This is done using a Template Haskell macro named declare, which is included in the library.

\[
\begin{align*}
declareByPhone &= \text{\$}\left(\text{declare } [\text{ByPhone}] 1\right) \\
declareByEmail &= \text{\$}\left(\text{declare } [\text{ByEmail}] 1\right) \\
declareNotAtAll &= \text{\$}\left(\text{declare } [\text{NotAtAll}] 0\right)
\end{align*}
\]

For each of the constructors, we provide the macro with its name and arity. The delimiters $(...)$ and $[...]$ are Template Haskell syntax, which the library user does not need to worry about.

The three fresh subject type combinators are used to turn forms with subject types (respectively) Phone, String and no subject type at all into forms with subject type Contact. Their type signatures are:

\[
\begin{align*}
declareByPhone :: \text{Monad } m \Rightarrow \\
(\text{Ref } m \text{ Phone } \rightarrow z) \rightarrow \text{Ref } m \text{ Contact } \rightarrow z \\
declareByEmail :: \text{Monad } m \Rightarrow \\
(\text{Ref } m \text{ String } \rightarrow z) \rightarrow \text{Ref } m \text{ Contact } \rightarrow z \\
declareNotAtAll :: \text{Monad } m \Rightarrow \\
\text{FForm } \text{win } m l \rightarrow \text{Ref } m \text{ Contact } \rightarrow \text{FForm } \text{win } m l
\end{align*}
\]

In the last type signature, the type \text{FForm \text{win \text{m \text{l}}}} plays the same role as \text{z} in the above signatures. The reason why it is more constrained is that \text{declareNotAtAll} appends its argument form with an invisible form for handling the \text{NotAtAll} value.

Using these subject type combinators, contactForm\text{\textsubscript{1}} can be defined as follows; we assume (phoneForm :: \text{Ref } m \text{ Phone } \rightarrow \text{FForm } \text{win } m \text{ Layout}) is defined somewhere else:

\[
\begin{align*}
\text{contactForm\textsubscript{1}} &= \text{radioGrid } [\text{byPhone, byEmail, byNothing}] \\
\text{byPhone} &= \text{declareByPhone } $ \lambda \text{phone} \rightarrow \\
& \quad \text{row}’5 [\text{label} “by phone :”, \text{phoneForm } \text{phone}] \\
\text{byEmail} &= \text{declareByEmail } $ \lambda \text{email} \rightarrow \\
& \quad \text{row}’5 [\text{label} “by email :”, \text{entry}’ [] \text{email}] \\
\text{byNothing} &= \text{declareNotAtAll } $ \\
& \quad \text{label} “do not contact me”
\end{align*}
\]

The new disjoint form combinator \text{radioGrid} arranges its list of subforms into a grid layout, with radio buttons to the left of them. Due to their subject type

\footnote{Template Haskell is a GHC compiler extension for meta-programming, i.e. programmatically manipulating a program at the syntactic level. The delimiters turn a meta-language expression into an object-language expression and vice versa. Both object language and meta-language are Haskell.}
combinators, the three subforms have the same subject type as the composite form \( \text{Contact} \), but each only ‘works’ for a particular data constructor. For example, the \( \text{byEmail} \) form only handles \( \text{ByEmail} \) values. This means that when \( \text{contactForm}_1 \) is run with an initial \( \text{ByEmail} \) value, the middle radio button is selected, and only the text field next to it receives an initial value. The other text field is left empty (or contains a default value, if the programmer has specified this in \( \text{phoneForm} \)). When the form is closed, every subform contains a final value with its particular \( \text{Contact} \) data constructor, but only one of them is promoted to \( \text{contactForm}_1 \)’s final value; this choice is determined by the radio button selected at that time.

What is the advantage of using the disjoint form combinator \( \text{radioGrid} \), apart from stylistic arguments? Consider the alternative case, in which the form in Figure 8.3 is defined as a conjunction of a \( \text{radioBox} \) (with an \( \text{Int} \) for three possible choices), a \( \text{phoneForm} \) and an \( \text{entry} \); its subject type would be \( (\text{Int}, \text{Phone}, \text{String}) \). At some later time, the interaction design department decides the form should rather look like Figure 8.4 or like Figure 8.5. Note that these forms still express exactly the same choice. However, when the form code is changed accordingly, its subject type would be \( (\text{Int}, \text{Int}, \text{Phone}, \text{String}) \) or \( (\text{Bool}, \text{Int}, \text{Phone}, \text{String}) \), and the code which handles the form data should also be altered.

If we use disjoint forms instead, the disjoint subject type can remain the same. In the code, we only need to add an extra \( \text{radioGrid} \) for the first case:

\[
\text{contactForm}_2 = \text{radioGrid} \\
\text{noContact} = \text{declareNotAtAll} \ \
\text{yesContact} = \text{radioGrid} \ [\text{byPhone}, \text{byEmail}]
\]

For the second case, we use another disjoint form combinator, namely \( \text{checkRow} \):

\[
\text{contactForm}_3 = \text{checkRow} \\
(\text{yes} \rightarrow \text{column} \ 5 \ [\text{label} \ “\text{Please contact me}\”, \text{yesContact yes}]) \\
(\text{declareNotAtAll} \ \text{noLayout})
\]

The functionality of these forms is still the same: they display a value of their subject type \( \text{Contact} \), and allow the user to change it into another value of that type.
Chapter 8. Disjoint Forms in Graphical User Interfaces

8.4 IMPLEMENTATION

Although the user of FunctionalForms does not notice a difference, apart from the new combinators and slightly altered Ref and FForm types, the implementation of the library has undergone substantial changes since its first version in [EAK05]. These allow for generalized forms, which may fail to consume an initial value (or produce a final value), and which can be joined with the disjoint form combinators radioGrid and checkRow. To construct these forms, the compositional functional references have also been generalized. Furthermore, the ‘heart’ of a form, which determines the communication with its environment, has been made explicit in a type RefLink. In order to deal with the new FForm type in Section 8.4.3 we will first discuss these Ref and RefLink types.

8.4.1 The Ref type

A reference value consists of two functions which are used to ‘refer to’ a t part of a – usually stateful – monad m:

\[
\text{data Ref } m \; t = \text{Ref} \{ \text{val :: } m \; t,
\text{app :: } (t \rightarrow m \; t) \rightarrow m () \}
\]

The val function retrieves the value of this particular part of the monadic state, whereas the app function updates it. For example, for a state of type \((t_1, t_2)\), the value referring to the \(t_1\) element would be:

\[
\text{reffst :: MonadState (t_1,t_2) m \Rightarrow Ref} \; m \; t_1
\text{reffst = Ref} \{ \text{val = do} \{ (x,y) \leftarrow \text{get}; \text{return } x \}
\text{, app = \lambda f \rightarrow do} \{ ~(x,y) \leftarrow \text{get}; x' \leftarrow f \; x; \text{put } (x',y) \} \}
\]

Note the lazy pattern match in the app function; it is useful when constructing a new state from scratch (i.e. the previous state contains \(\bot\)).

A reference to the value in a Just data constructor (from the well-known Maybe type) can be defined in a very similar way:

\[
\text{refffromJust :: MonadState (Maybe t) m \Rightarrow Ref} \; m \; t
\text{refffromJust = Ref}
\{ \text{val = do} \{ \text{Just } x \leftarrow \text{get}; \text{return } x \}
\text{, app = \lambda f \rightarrow do} \{ ~(\text{Just } x) \leftarrow \text{get}; x' \leftarrow f \; x; \text{put } (\text{Just } x') \} \}
\]

This reference value may seem ill-defined because it can ‘dangle’: when the monadic state contains Nothing, it does not refer to anything. However, this situation can be detected using monadic error-handling, and the control flow can be adapted. We will show how this is done in Section 8.4.2 when we join two RefLinks.
The operator \( \circ \) composes two reference values, taking the referred part of the second value’s state as the state of the first value. For example, the following value refers to the first element of a pair within a Just value:

\[
\text{reffst} \circ \text{reffromJust} :: \text{MonadState} (\text{Maybe} (t_1, t_2)) m \Rightarrow \text{Ref} \, m \, t_1
\]

The composition is performed by applying a monad transformer to the monad of the second reference value. This ‘adds state’ to this monad, on which the first reference value can act. Meanwhile, properties of the original monad such as IO ability or error handling are preserved.

\[
\bullet :: \text{Monad} \, m \Rightarrow \text{Ref} \, (\text{StateT} \, cx \, m) \, t \rightarrow \text{Ref} \, m \, cx \rightarrow \text{Ref} \, m \, t
\]

\[
\text{w} \bullet \text{v} = \text{Ref}
\]

\[
\begin{array}{l}
\{ \text{val} = \text{val} \, v \Rightarrow \text{evalStateT} (\text{val} \, w) \\
\hspace{1cm}, \text{app} = \lambda f \rightarrow (\text{app} \, v) \, \text{execStateT} \, (\text{app} \, w) \, \text{lift} \, .f
\}
\end{array}
\]

This operator is used to define subject type combinators like:

\[
declare2 :: \text{Monad} \, m \Rightarrow (\text{Ref} \, (\text{StateT} \, cx \, m) \, t) \rightarrow \text{Ref} \, m \, cx \rightarrow z
\]

\[
declare2 \, \text{refsToForm} \, \text{refP} = \text{refsToForm} \, (\text{reffst} \circ \text{refP}, \text{refsnd} \circ \text{refP})
\]

\[
declare\text{Just} :: \text{Monad} \, m \Rightarrow (\text{Ref} \, m \, a \rightarrow z) \rightarrow \text{Ref} \, m \, (\text{Maybe} \, a) \rightarrow z
\]

\[
declare\text{Just} \, \text{refToForm} \, \text{refMaybe} = \text{refToForm} \, (\text{reffromJust} \circ \text{refMaybe})
\]

These subject type combinators all follow the same pattern. This pattern is captured in the Template Haskell macro \text{declare}, so definitions like the two above do not have to be handwritten for every data constructor.

### 8.4.2 The RefLink type

The heart of an atomic form consists of a link between two reference values. The first is of type \( \text{Ref} \, m \, t \), and relates the subject type \( t \) of this form to that of the topmost form (the state type in \( m \)). This is the reference value that is explicitly provided by the library user, e.g. in the expression \text{entry}'[] \text{ref}_m. The second reference value is implicit in every atomic form; it is of type \( \text{Ref} \, \text{IO} \, t \), and relates this form’s subject type to a part of the IO state. It is constructed from \text{wxHaskell’s get} and \text{set} functions for the control’s main attribute (e.g. \text{text} on an \text{entry} control).

In the terminology of the well-known model–view–(controller) paradigm [KP88], the reference values refer to a part of the topmost form’s model and a part of its view, respectively. Joining them in a RefLink means linking those parts to each other: the \text{val} output from the first reference is used as \text{app} input for the second, and vice versa. Thus, two operations are obtained which both enforce consistency between model and view:
• The `update` operation copies the value from the form’s model to its view. This is used to show the form’s initial value.

• The `propagate` operation copies the value from the form’s view to its model. This is used to read the form’s final value.

In monadic terms, the `update` operation is a read action in the `m` monad producing a write action in the `IO` monad. Conversely, the `propagate` operation is a read action in the `IO` monad producing a write action in the `m` monad. This results in the following type for `RefLink` (where the `n` monad abstracts from `IO`):

```haskell
data RefLink m n = RefLink
  { update :: m (n ()),
    propagate :: n (m ())
  }
```

The function `refLink` connects the two references to create such a `RefLink`. For the `update` function, first an input `v` is retrieved from the `m` reference. Then, a constant function `const (return v)` is applied to the corresponding part in the `n` monad using the `n` reference; this action in the `n` monad is returned in the `m` monad. For the `propagate` function, the roles of `m` and `n` are reversed:

```haskell
refLink :: (Monad m, Monad n) ⇒ Ref m a → Ref n a → RefLink m n
refLink ref_m ref_n = RefLink
  { update = (val ref_m) >>= return . (app ref_n) . const . return,
    propagate = (val ref_n) >>= return . (app ref_m) . const . return
  }
```

When two forms are joined, their `RefLinks` are combined into a new `RefLink`. Usually, the intention is that the joint `update` performs both component `updates`, and likewise for the `propagates`. We consider this to be the ‘default’ operator on `RefLink`. In order to meet the `MonadWriter` interface (see Section 8.4.3), we encode it using the `Monoid` class:

```haskell
instance (Monad m, Monad n) ⇒ Monoid (RefLink m n)
  where mempty = RefLink
    { update = return $ return (),
      propagate = return $ return ()
    }

  rl1 `mappend` rl2 = RefLink
    { update = liftM2 (≫) (update rl1) (update rl2),
      propagate = liftM2 (≫) (propagate rl1) (propagate rl2)
    }
```

For disjoint forms, the two `RefLinks` should be joined in an alternative way. In this situation, they share one part of the model part, which is a disjoint union (e.g. the subject type of both forms is `Either a b`). Meanwhile, they refer to different parts of the view (which contains controls for both `a` and `b`). Hence, the two
RefLinks connect independent parts of the view state space to ‘competing’ parts of the model state space. Instead of performing both update (propagate) operations, only one can (should) be performed.

We obtain this behaviour by using the mplus operator of the model monad m; therefore, this should be an instance of MonadPlus. The joint update will then (dynamically) choose between the first component update or the second – and likewise for propagate. Hence, we define an alternative monoid on the RefLink domain. By using a different representation for the RefLink type, the Monoid class can again be used for this:

```haskell
newtype RefLinkPlusM m n = RefLinkPlusM { pm :: RefLink m n }
instance (MonadPlus m, Monad n) ⇒ Monoid (RefLinkPlusM m n) where
  mempty = RefLinkPlusM $ RefLink
    { update = mzero,
      propagate = return mzero
  }
  rl1 `mappend` rl2 = RefLinkPlusM $ RefLink
    { update = (update $ pm rl1) `mplus` (update $ pm rl2),
      propagate = liftM2 mplus (propagate $ pm rl1) (propagate $ pm rl2)
  }
```

The exact semantics of mplus depend on the monad m. In practice, we use an error-handling state monad ErrorT e (State a). This means that the first argument of mplus is always tried first; if it fails, the second argument is tried. When disjoint forms are used correctly, the alternatives are mutually exclusive, so this order is irrelevant.

### 8.4.3 The FForm type

A form is a value of the following type:

```haskell
newtype FForm win m a = FForm
  { runFForm :: Window win → IO (a, RefLink m IO) }
```

It contains three pieces of information:

1. An IO action which creates the form’s controls. This action depends on a pointer to a parent window of type Window win, in which they are created.
2. A RefLink used to update the values in the controls from, and propagate them to, the form’s model.
3. Additional information of type a; usually layout information of type Layout (defined by the wxHaskell library).

The FForm type can be used as a monad, which binds the additional (layout) information, reads the window pointer, executes the control creation functions, and writes a RefLink.
instance (Monad m) ⇒ Monad (FForm win m)
instance (Monad m) ⇒ MonadReader (Window win) (FForm win m)
instance (Monad m) ⇒ MonadIO (FForm win m)
instance (Monad m) ⇒ MonadWriter (RefLink m IO) (FForm win m)

So, \( \form_1 \gg f \) means:

- \( f \) is applied to the additional information from \( \form_1 \), producing (let’s call it) \( \form_2 \).
- The window pointer passed to \( \form_1 \gg f \) is passed to \( \form_1 \) and \( \form_2 \).
- The IO actions in \( \form_1 \) and \( \form_2 \) are sequenced.
- The RefLink in \( \form_2 \) is joined to the RefLink in \( \form_1 \) using the ‘default’ mappend operator.

Furthermore, functions like ask (extract the window pointer), liftIO (insert an IO action at form creation time) and tell (insert a RefLink) are implemented for the FForm monad (being an instance of MonadReader, MonadIO and MonadWriter).

We stated in Section 8.4.2 that in order to combine two forms in a disjoint way, the RefLinkPlusM monoid should be used, which dynamically performs one of the update/propagate operations. Meanwhile, both forms should be shown: at form creation time, both IO actions should be performed, and both layout values are used. Therefore, we have implemented alternative bind and unit operators for forms: \( \gg \pm \) and \( \text{return}^0 \). They are similar to \( \gg = \) and \( \text{return} \) in every respect, except that they use the RefLinkPlusM monoid.

\[
\text{return}^0 :: \text{MonadPlus} m ⇒ a → FForm win m a \\
(\gg \pm) :: \text{MonadPlus} m ⇒ FForm win m a → (a → FForm win m b) → FForm win m b
\]

These operators are at the core of the disjoint form combinators radioGrid and checkRow, whose implementation is discussed in the next section.

### 8.4.4 Disjoint form combinators

A naïve disjoint form combinator would be:

\[
\text{refToForm}_1 \text{'or'} \text{refToForm}_2 = λref → \\
\text{refToForm}_1 \text{ref} \gg \pm λ\text{lay}_1 → \\
\text{refToForm}_2 \text{ref} \gg \pm λ\text{lay}_2 → \\
\text{return}^0$ column 5 [\text{lay}_1,\text{lay}_2]
\]

The composite form shows both forms, while the composite update function performs only one of the component update functions – the first one that succeeds. The same goes for the composite propagate operation. However, the form’s user
has no means whatsoever to discover which \textit{update} has been performed, or to influence which \textit{propagate} to perform\footnote{Both \textit{propagate} operations will succeed when performed. Due to the \textit{mplus} semantics, the first one will always be selected.}.

The \texttt{radioGrid} combinator does provide these functions: both are fulfilled by the radio buttons. When a subform’s \textit{update} is performed, the system selects the radio button in front of it. Conversely, the form’s \textit{propagate} is only performed if the radio button in front of it \textit{is} indeed selected (the user influences this during the form’s lifetime).

The nice thing is that we can express this behaviour quite elegantly in terms of \texttt{RefLink} operations. We show this by defining the somewhat simpler disjoint form combinator \texttt{alt}, which is a specialisation of \texttt{radioGrid}; it joins exactly two forms (denoted \texttt{refToForm}_1 and \texttt{refToForm}_2). Assume that we can create a two-button radio group, returning a reference value \texttt{refRadio :: Ref IO Int} to its current selection, which can take values \{0, 1\}. Now we can define a \texttt{RefLink} value:

\[
rl_1 = \text{refLink ref}0 \text{refRadio}
\]

\texttt{where} \texttt{ref}0 = \texttt{Ref}

\[
\begin{aligned}
\{ & \text{val} = \text{return } 0 \\
& ; \text{app} = \lambda f \rightarrow \text{do } \{ 0 \leftarrow f \; 0 ; \text{return } () \} \\
\}
\end{aligned}
\]

In other words: we link \texttt{refRadio} to a reference ‘to the unchangeable number 0’, whose \texttt{val} is always 0, and whose \texttt{app} function only succeeds when the result of the function application is 0. This has the effect that the \textit{update} operation in \texttt{rl}_1 always selects the topmost radio button (and succeeds), while its \textit{propagate} operation only succeeds when this radio button is selected\footnote{Note that this \texttt{RefLink} does not use any model state!}.

We then lift \texttt{rl}_1 into a form, and join it with the first subform \texttt{form}_1 using \texttt{≫}, which utilizes the default (conjunctive) \texttt{mappend} operator:

\[
tell rl_1 \gg refToForm_1 ref
\]

This form has the desired properties: with an \textit{update}, the radio button is only selected if the value in \texttt{form}_1 can be updated, and with a \textit{propagate}, the value in \texttt{form}_1 is only propagated if the radio button is selected. We define and use \texttt{rl}_2 in a similar way, producing a second form. We finish the \texttt{alt} combinator by joining both forms with \texttt{≫±}:

\[
\begin{aligned}
\text{refToForm}_1 \text{‘alt’ refToForm}_2 &= \lambda ref \rightarrow \\
liftIO \ldots & \gg\gg (\text{laybutton}_1, \text{laybutton}_2, \text{refRadio}) \rightarrow \\
\text{let} & \\
r_1 &= \ldots \text{ — see above} \\
r_2 &= \ldots \\
\text{in} \\
(\text{tell } r_1 \gg \text{refToForm}_1 ref) \gg\gg \lambda \text{layform}_1 \rightarrow \\
(\text{tell } r_2 \gg \text{refToForm}_2 ref) \gg\gg \lambda \text{layform}_2 \rightarrow \\
\text{return}^0 \; $ \; \text{grid } 5 \; 5 \; \{[\text{laybutton}_1, \text{layform}_1], [\text{laybutton}_2, \text{layform}_2]\}
\end{aligned}
\]

\[
\text{grid } 5 \; 5 \; [[\text{laybutton}_1, \text{layform}_1], [\text{laybutton}_2, \text{layform}_2]]
\]
What we have omitted in the second line goes into too much implementation detail; it is an IO action which creates the radio buttons, and returns the layout values \texttt{laybutton}_1 and \texttt{laybutton}_2, as well as \texttt{refRadio}.

The \texttt{radioGrid} combinator is a straightforward generalization of \texttt{alt} for lists. The \texttt{checkRow} combinator is also very much like \texttt{alt}, but does not show its second argument form. However, it \textit{does} use its \texttt{RefLink}.

### 8.5 SAFETY

The flexibility provided by compositional functional references has a downside: by omitting reference values, duplicating them, or using them in the wrong places, forms with strange behaviour can be constructed. We give some examples:

1. \texttt{declare2 \lambda(a,b) \rightarrow entry' [] a}
   
   This form never shows or changes the second element of its subject type.

2. \texttt{\lambda a \rightarrow row' 5 [entry' [] a, entry' [] a]}
   
   This form shows its value twice, and only propagates the new value in the control on the right.

3. \texttt{declareJust $ entry' []}
   
   This form is only updated if its model contains a \texttt{Just x} value (actually, this is the desired behaviour if the form is part of a disjoint form). If it does not, all forms in the same alternative of the surrounding ‘disjoint clause’ are prevented from being updated (normally, there should be none).

4. \texttt{radioGrid [entry' [], entry' []]}
   
   This form will always put its value in the upper entry control. However, it will propagate values from whichever entry control has its radio button selected.

To prevent the construction of these forms, the programmer can follow some rules such as:

- Every declared reference should be used exactly once.
- Every data constructor of a form’s subject type should be declared exactly once.
- References declared outside a disjoint form must not be used inside it.

Of course, it would be better if these rules would be enforced automatically, e.g. by the type system. Future research should formalize these rules.

---

\footnote{Its \texttt{RefLink} is joined with a \texttt{refLink} between the value \texttt{False} and the checkbox value.}
8.6 RELATED WORK

As far as we know, the idea of explicitly using a radio button grid to combine forms in a disjoint way is new. The fact that some radio buttons make other elements (ir)relevant is recognized, but existing declarative (web) form languages have to go out of their way to specify this. In XForms \cite{XFo}, it is accomplished by providing an element’s relevant property with a Boolean expression that includes an XPath pointer to the radio button choice. In WASH/CGI \cite{Thi02}, the programmer builds a decision tree (see \cite{Thi03}) to express which data to use when a certain radio button is selected.

A simple disjoint form combinator is already introduced in the thesis \cite{Eve04} from which FunctionalForms originated. However, this combinator always joins exactly two alternative forms. If the subject types of the top form and bottom form are $t_1$ and $t_2$, respectively, then the subject type of the composite form is always $\text{Either } t_1 t_2$. In other words, logic and layout are not separated like they are presently.

Compositional references were introduced by Kagawa \cite{Kag97} as a means to compose mutable data structures such as arrays. In our previous work \cite{Eve04,EAK05} we used them in a more simple form and with a different goal: to conceptually separate a form’s subject type and its layout.

Closely related to compositional references are lenses \cite{FGMPS04}, which are also pairs of accessor and modifier functions. While our approach uses a lot of ‘little’ references throughout the program, lenses are combined into a big lens which is the program; this program specifies a bidirectional transformation between model and view.

8.7 CONCLUSIONS AND FUTURE WORK

In this paper, we have identified two patterns for composing forms that edit values of a disjoint union type. The first pattern involves a list of radio buttons, and the second involves a check box. To support these patterns in the FunctionalForms library, we have introduced several new combinators.

These patterns illustrate a novel view, in which a form itself can be seen as ‘disjoint’. To demonstrate the fertility of this view, we have shown that these disjoint forms exhibit a cleaner separation between logic and user interface. This makes them more flexible.

However, this flexibility comes at a price: the construction of forms with unwanted behaviour is possible. Methods for preventing this have yet to be researched.

As a further enhancement to FunctionalForms, defining forms for values of a custom Haskell type is made easier, using a Template Haskell macro. This brings the library closer to real-life use. The library version discussed in this paper will shortly be available for download\footnote{http://www.sandr.dds.nl/FunctionalForms}. 

\hspace{1cm}
We hope to further develop the approach of programming with reference values. We believe that it can be used to construct a far wider range of interfaces in a declarative way.

REFERENCES


Chapter 9

A Graphic Functional-Dataflow Language

Silvia Clerici¹, Cristina Zoltan¹

Abstract: NiMo (Nets in Motion) is a visual environment aimed to support totally graphic programming in Data Flow style, with a strong functional inspiration. It enables the user to describe solutions of growing complexity using a small set of graphic primitives. The NiMo language allows multiple output channels and adds graphic representation to deal with functional concepts like higher order, partial application, laziness, polymorphism and type inference. The environment is based on a graph transformation system, by means of which execution can be done step-by-step. NiMo intends to be a workbench for editing, debugging, executing and experimenting. Nets are built and run using a uniform conceptual framework. The net to be initially executed is exactly the one drawn by the user. It evolves showing all transformations of data and processes. The most distinguishing characteristic of NiMo is allowing the user to execute incomplete programs and to change the execution state (data, processes and/or control) in an interactive way.

9.1 INTRODUCTION

Functional languages such as Miranda [Tur85] or Haskell [HJW⁺92, Has04] have powerful primitives for building algorithms in a simple way. Higher order functions are program schemes that allow building software pieces from higher levels of abstraction. Solutions are obtained by the composition of built-in blocks of well-known semantics. Reusability is promoted by defining polymorphic higher

¹Dept. Llenguatges i Sistemes Informàtics Universitat Politècnica de Catalunya Barcelona, Spain; E-mail: {silvia,zoltan}@lsi.upc.edu
order functions that extend the predefined schemes with more specialized ones.

Data flow is a simple and powerful computational model [Den74] that has inspired several visual programming languages. In lazy functional programming, data flow representations visualize the program as a net of communicating processes where functions are processes and channels are (usually infinite) lists. During the last years the interest for visual modelling and programming has progressively grown, giving rise to a wide variety of visual languages and environments [JHM04]. In particular there are a certain number of them close to functional programming or data flow paradigm: Prograph [CGP89], Visual Haskell [Ree94], VFPE [Kel02], Vital [Han02]. They were developed with different objectives and therefore the term ‘Visual’ has different meanings: graphical edition, visual representations of textual programs, data structures animation, graphical display of the internal representation, visualization of how some part of it evolves during execution, etc. Some of these languages share certain characteristics with NiMo but their specific objectives and the way they are implemented are fairly different.

The main goal in NiMo is to provide totally graphic edition, debugging, execution and experimentation. The source code is a graph, which is directly executable. It is not a graphic representation of the code of a given textual language as it is the case in several visual languages, for instance in Visual Haskell. Although close enough to lazy functional languages NiMo follows a functional-data-flow mix model, and there is no need for the user to write textual code at all. On the other hand, although the graph could be easily translated into a lazy functional language as Miranda or Haskell, this is not the case either. The interpreter acts directly upon the source code, without any kind of translation, by means of a graph transformation system. The execution model in NiMo could be regarded as the visualization process of a graph operational semantics for an equivalent textual program, where the state model is a graph; during execution all the intermediate states are shown, not only the final one.

The NiMo environment, which is described in Section 9.3, is based on a graph transformation system [Roz97]. This means that NiMo implementation is also graphical. It consists of two kernel graph grammars; one of them acts as an interactive syntax directed editor while the other one is the interpreter, which acts directly on the source code. Execution can be done in free mode or step-by-step also allowing incomplete programs to be executed and completed as needed.

There are several tracers for functional programs that also share some functionalities with our work. In particular for the Haskell language we can mention HsDebug [EJ03] for GHC [Com04], HOOD which is a small post-mortem debugger, being enhanced by GHOOD [GHO04]. GHOOD has a very nice interface for visualization of selected elements of the program. All of them require signaling the functions that are to be observed. The main difference with them is that NiMo acts as an online tracer. All the net activity is shown by default; therefore the marking should be for the processes of which we do not want to see the internals. Step-by-step execution allows online debugging and also several kinds of experiments on the running net, like relaxing laziness for some processes, bounding the execution to a particular subnet, analyzing channel population, testing the need of
forcing synchronization, etc.

A running prototype has been implemented using AGG [ERT99, TB04] as the graph transformation system. It is described in Section 9.4 together with its possible extensions.

In the next section the main characteristics of the language are outlined and illustrated by means of simple examples of its use.

9.2 OUTLINE OF THE NIMO LANGUAGE

The language has been designed to support totally graphic programming in Data Flow style, with a strong functional inspiration. It is based on the classical process network representations for lazy functional programs. A small set of graphic primitives allows representing and handling functional concepts like higher order, partial application, laziness, polymorphism, and type inference. The program is a graph drawn by the user, which evolves during execution showing every transformation of data and processes. The most peculiar characteristic of NiMo is that the source code is both the object code and the initial state at the same time. This feature, added to the possibility of executing (may be incomplete) programs step-by-step, enables the user to change the state (data, processes and/or control) at any point. This section outlines the main features of the language and the functional-data-flow mix model it supports, starting with the usual process network representations in which NiMo was inspired.

9.2.1 Data flow representation for lazy functional programs

In the basic data flow model processes are visualized as boxes (nodes), and arrows represent channels where data streams flow from one process to other. The control is given by the flow of data; there are no additional control structures. Data-flow representations have shown to be a useful conceptual mechanism to visualize the behaviour of a certain kind of lazy functional programs. The data flow metaphor

![Fibonacci process network](image)

**FIGURE 9.1.** Fibonacci process network

associates functions to single output processes and channels to (usually infinite) lists. Net architecture shows in a bi-dimensional way the chains of function compositions and exhibits the implicit parallelism. Back arrows give an insight of the ‘recurrence laws’, i.e. how new results are obtained from already calculated ones.

The paradigmatic example of such correspondence is Fibonacci. Its process
network representation, present in many textbooks on lazy functional programming, and the graphic execution model that the net animation suggests, were the start point for the NiMo language design. Figure 9.1 shows the net corresponding to the Haskell code

\[\text{fibo} = 0 : \text{rest} \text{ where rest} = 1 : \text{zipWith (+) fibo rest}\]

### 9.2.2 The graphic elements

The graphic syntax in NiMo is very close to the one above. Figure 9.2 shows the Fibonacci net in NiMo.

![Fibonacci process network in NiMo](image)

Except for a few new graphic symbols (hexagons, diamonds, and big-arrows shown in Figure 9.3), the rest of the elements and their correspondence to the textual code are the usual ones:

- Rectangles for functions viewed as processes.
- Circles (or ellipses when necessary) for constant values.
- Arrows for function parameters and result, avoiding the use of the identifiers needed in textual code (fibo and rest in this example).
- Horizontal arrows to represent channels and black dots for channel duplicators (double occurrence of an identifier in the textual code definition).
- Vertical arrows for those function parameters that are not channels in the data flow sense of flowing data streams.

The diamond symbol is a connector that ties arrows together and carries information about activation or evaluation state. A diamond with 0 precedes a non-activated process or a non-evaluated value. Values in a channel are always preceded by a diamond with code 1. Hexagons carry type information. Data are type-value-entities (tve), where values (evaluated or not) are tied to its type (hexagon) through a diamond. Code 2 in the diamond indicates that the value is an atomic constant. Fully evaluated values (values in normal form) are tve’s having only hexagons, circles and diamonds. Non-fully evaluated values are tve’s containing
at least one rectangle (a non-evaluated function), see Figure 9.6. Formal parameters are represented by incomplete tve’s; i.e. having big arrows (as shown in Figure 9.3). In fact, big arrows represent incompleteness in a more general sense; they are open connections that can be bound later. Since execution can be done step-by-step, incomplete programs can be executed and completed by need.

NiMo allows full higher order parameterization and partial application. Process parameters can also be parameterized; i.e. not only function names as +, but also partially applied functions (a box with big-arrows in some of its inputs) can be a valid parameter for another process, see the net quickSrec in Figure 9.7. Furthermore, processes in NiMo can have multiple outputs, but only single output processes can be a valid parameter. In other words, functions can be viewed as processes when applied, but multiple output processes cannot be viewed as functions.

9.2.3 The type system

Atomic types are integers (I), reals (R), booleans (B), strings (S). Circles of different colours represent constant values of each type.

Structured types are lists (L), channels (C), tuples (T), and functions (F) all being polymorphic. Channels and lists are homogeneous and unbounded. They correspond to the data flow and functional view of lists and therefore they are mutually convertible.

In the current version user defined types are not supported. On the other hand, there is only one function type name F standing for the type $a \rightarrow b$, only one list type name L for lists of any kind of elements, and the only polymorphic type identifier is ?. The type List of integer, for instance, corresponds to a list-tve (a tve with L in its hexagon) with integer-tve’s as elements of its (structured) value. Therefore only lists having a first concrete typed element show they are concrete; an empty list or a non-evaluated one (not in w.h.n.f.) is, in principle, polymorphic. The same happens with channels. As a consequence, static type inference is partial, which means that polymorphic processes could be non-correctly instantiated and therefore non type-safe nets could be executed. Type inference and checking at runtime is therefore needed (see Section 9.3.1 for further discussion). In the NiMo next version (under preparation) full static inference will be possible.

9.2.4 Basic processes and nets

As said before, NiMo language was designed to support the data flow process network programming style. The system provides a repertory of powerful basic processes including equivalent versions for most functions in Haskell prelude. It also includes additional higher order processes that facilitate the static process network style, like a generalized zipWith-merge, a flip-flop-merge, or a fixpoint process. But with some of the Haskell functions (like splitAt, span, or break) the equivalence is not exact because processes in NiMo can have more than one output channel, which can be independently activated (like the process splitCond in
Process network solutions obtained using these primitives as basic processes avoid the use of tuples and projections or ‘symmetric duplication’ of processes, which are needed to simulate two output branches in Haskell. In fact, tuples are only used in NiMo when dealing with truly Cartesian Product elements, and could be simulated by the multiple outputs of a process. On the other hand, additional graphic syntax for conditional, guards or case expressions is not either really necessary: an if-then-else higher order process suffices for handling all the conditional situations. The idea is that casuistic and pattern decomposition are encapsulated within the basic processes behaviour. Basic processes are powerful enough to handle stream processing in the data flow sense; only the simplest user functions (i.e. not involving control flow) should need conditionals. This restriction enforces a static network solution style, without explicit recursivity in processes; recurrence is done on channels (feed back arrows). In this kind of solutions the total number of processes is statically bounded. During execution the net structure is similar to the original one, regarding the processes and their interconnections, only new values are dynamically generated in channels (as in the case of the net prefixes, explained in the next sections). However, as it is discussed in Section 9.2.6, dynamic nets can also be represented in NiMo. In this case, a graphic syntax for channel patterns would be helpful to define recursive nets.

Anyhow, the user builds solutions combining processes that can be basic or net themselves. Every new net can be named to be used by another one, allowing incremental net complexity up to any arbitrary degree, and nets can also be parameterized. Figure 9.3 shows the net that generalizes the Fibonacci algorithm to obtain successions where the first and second elements, and the function to apply to the two previous values, are formal parameters. Some useful parameterized nets are provided in a library that can be extended with user-defined nets.

**FIGURE 9.3. Generalized Fibonacci process network**

### 9.2.5 The execution model

In functional languages the execution model strongly depends on the overall evaluation policy. In general, it can be lazy or eager (although efficiency reasons can relax the ‘purity’ of the evaluation order). In lazy languages there are primitives to annotate a sub expression to be eagerly evaluated; it is the only way the user can
explicitly modify the control. On the contrary, in NiMo each subnet is explicitly preceded by its evaluation state, and the user can set an initial activation or change the current activation state of any process during execution. Nevertheless, the default policy is lazy; basic processes have a lazy behaviour, activation propagates by need and persists until an ‘sufficiently evaluated’ result is obtained. Therefore, provided that the net initial marking only activates the output channels, and that all its sub-nets have a lazy behaviour, the net itself will have a lazy behaviour. However other possible markings could also respect laziness, as it is discussed in Section 9.2.6 for the net in Figure 9.9.

In the basic data flow model, control is data driven, i.e. processes act by data availability. In the mix model, laziness means that processes only produce results under demand, i.e. the control flow follows a demand-driven policy (the only exception to this rule appears when simplification keeps semantics). Therefore, executing the Fibonacci net requires creating a demand on its output channel, i.e. setting 1 in its preceding diamond. The first process ‘:’ acts putting its parameter on its output channel and then disappears; the resulting net is shown in 9.4.

This first value 0 is then shown as the first result and it is also duplicated, being so available in the first input channel of zipWith. Value entities in a channel are always preceded by a diamond with code 1. Figures 9.5 and 9.6 show the net before and after the third value is calculated. In the general case, an activated
process can demand non-available values to its provider processes, which are activated ‘in parallel’ (all that needed, not only the ‘leftmost one’, as in sequential lazy languages) and, in turn, activate their own providers. Then, process activation can propagate and therefore several processes could work simultaneously in different regions of the net. In the current implementation, this means that whenever two processes are activated, the application order is non deterministic, as it is further discussed in Section 9.3. In NiMo most basic processes only act when their preceding diamonds have code 1, (as it was the case for zipWith). However, due to efficiency reasons, some processes could act without being activated (e.g. + whenever its two operands are available, or map with an empty input channel).

Once activated, a basic process has a lazy behaviour; it only propagates activation in the input channels it needs to produce its result, i.e. to put a next value on every activated output channel; then it becomes deactivated (all its preceding diamond have a 0-value). In other words, processes act until a w.h.n.f. is obtained in each activated output channel.

In order to allow a process to be continuously activated, NiMo provides a basic process force, which generates a continuous demand on its input channel and also a process force-eval that, in addition, completely evaluates its input. Besides, as execution can be done step-by-step, the user can activate/deactivate any process, by simply changing the code of its preceding diamond. This facility opens a wide range of experimental possibilities not present in the offline style debuggers because some regions of the net can be tested in an isolated manner at any point of the execution. Moreover, relaxing laziness online allows to explore possible optimizations and to analyze synchronization issues. It also enables parallel execution of subnets that, following the default lazy policy, should remain inactive.

### 9.2.6 Drawing dynamic nets

As we have pointed out in Section 9.2.4 static net programming style promotes the intensive use of built-in recursive patterns, and therefore channel pattern is not really needed. On the contrary, to draw dynamic nets it is convenient to have a more specialized syntax than the generic if-then-else basic process. Fig-

**FIGURE 9.6.** Fibonacci third value produced
Figure 9.7 shows the pair of (mutually recursive) nets for the quicksort algorithm. The net `quickSort` shows the graphic syntax for channel patterns, where upper or lower processes enclosed between triangles will be applied depending respectively whether the channel is empty or not. In execution, the sub graph between diamonds is replaced by one of the two processes. In this example `quickSrec` deals with the non empty case; afterwards it is replaced by its net which contains two new processes `quickSort`. This indirect recursion is necessary whenever the end of the recursion is given by a channel pattern; otherwise (for instance in the dynamical version of the Eratosthenes sieve), a single recursive net is enough. The basic process `splitCond` in the net `quickSrec` can be simulated in Haskell by a pair of symmetric filters giving both partitions of the input channel according to the condition and its negation. The big arrow in the `>=` process corresponds to the parameter of the `(a >=)` section in the Haskell code, being `a`, the first value in the input channel. The equivalent Haskell code for both nets is the following:

```haskell
quickSort [] = []
quickSort (a:x) = quickSrec(a:x)
quickSrec(a:x) = quickSort(filter(a>=)x)++
               (a:quickSort(filter(a<)x))
```

Let us note that the initial activation marking (diamonds preceding processes) of the net `quickSrec` preserves laziness. The sequence of diamonds with code 1 is exactly the one that would result by propagating the activation from the leftmost diamond (the one preceding the process `++`). According to this, the lower `quickSort` process in the net will not be activated until the upper one completes its task (because `++` first consumes completely its first input channel). A possible experiment is to activate also the second `quickSort`, allowing that both processes can act at the same time. This action should also affect the distribution of channel
population. In general, trying different markings on the net dynamically allows, for instance, to extend the implicit parallelism in order to speed-up the system behaviour in a further parallel implementation. Also, both the full vision of the state and the step-by-step execution enable the user to experiment even by changing values online.

Execution of both quicksort algorithm versions and other examples can be found in [NiM05].

9.3 THE NIMO ENVIRONMENT

NiMo has been developed with the goal of being a workbench for graphically editing, debugging and executing process network solutions. Nets are built and executed using a uniform conceptual framework. As in Functional Languages, where everything is an expression (data, processes, and also the program being executed), in NiMo everything is a graph. Graph grammars and graph transformation systems are powerful tools to deal with graphs. The NiMo environment is based on a graph transformation system that allows rule application to be done step-by-step or in free mode. It has two kernel attributed graph grammars: Construction Grammar (CGG), and Execution Grammar (EGG). EGG is the interpreter, and also gives the complete semantic definition of the language, while CGG acts as an interactive syntax directed editor that checks a first level of type-consistency.

9.3.1 Editing nets

In functional languages, the program is usually edited in a regular text editor and the whole text is compiled. The type inference system deals with the complete internal representation. On the contrary, in NiMo partial type checking and inference are made step-by-step, and locally, during the net edition. On the other hand, Graph Grammar based syntax checkers use techniques for ‘scanning’ a graph to verify the abstract syntax graph is not violated. In other cases the technique is checking if the ‘graph can be parsed’. The first approach is incomplete and the second is not step-by-step. In NiMo nets are constructed using the grammar CGG, its rules can assure a valid graph at each construction step.

![FIGURE 9.8. Interface rule for zipWith](image)

Every basic process or net in the system has an interface rule giving its type
declaration (input and output channels and parameters types). For example, the interface rule for the zipWith process is shown in Figure 9.8. During the net edition, adding a new process (basic or net) requires to create a circle, and to fill it in with the process name. The interface rule creates a template with the process graphic signature in the working graph. The new output big-arrows can be connected with the existing input ones, provided that their types are compatible. A set of rules handles partial type checking and inference. Partial means that in the presence of two polymorphic types (big-arrows having a non concrete type identifier in the hexagon) a non-safe connection could be allowed. For instance a first zipWith that (according to its functional parameter once it is connected) requires a first input channel of integer values, could be connected with a second zipWith that, according to its functional parameter, returns a boolean channel. This type inconsistence will be detected in execution when the first functional parameter is applied.

Once a new net is built, it can be directly executed or else be given a name to be further used. This action generates two new rules, an interface rule and an expansion rule like the one in Figure 9.9. The interface rule allows to construct other nets that have it as a process. It is also used in execution to allow higher order parameterization as described in Section 9.3.2. The expansion rule will be added to EGG to be used while executing a net having a box with the same name. When this process is activated, its whole net (the one on the right of the expansion rule) replaces the box that already has the necessary connections. Figure 9.9 shows the expansion rule for the net prefixes, which generates the list of all the initial sub-lists of a given list. This is one of the useful nets provided in the library.

![Expansion rule for the net prefixes](image)

**FIGURE 9.9.** Expansion rule for the net prefixes

### 9.3.2 Executing nets

As stated before, one of the most distinguishing characteristics of NiMo is that source code is not transformed to start execution; it is the object code itself. At the beginning of the execution the net is exactly the one drawn by the user, and EGG acts directly on it. In execution the net shrinks and expands following the evaluation needs. Diamond nodes keep track of process activation state or evaluation degree for values. Therefore, rules are applicable only on active regions of the net. If more than one process is active, rule application, being non-deterministic,
gives a good simulation (the visual effect) of concurrency.

Along with this, the graph transformation system not only provides mechanisms for interrupting execution, running the net in step-by-step mode, and undoing transformation steps but also allows modifying the net at any stage and resuming execution thereafter.

According to the demand-driven policy, the initial activation of the net is done by activating its output channels (although a different initial marking could be set). From then on, activation is propagated by need, in a waterfall mode. If the first process in a channel is not a basic one, it is expanded, and so on until a basic process can act. Once activated it is responsible for activating its provider processes. Basic process rules give the operative basis for the graphic transformation and moving of data through channels. The set of rules defining the behaviour of a given basic process is analogous to the equations defining a function in Haskell. Different configurations in its adjacent nodes give the patterns.

Figure 9.10 shows the three rules defining the behaviour of the basic process postfix (which adds an element at the end of a list). The first rule activates its provider process, since a 0-value in the input channel diamond indicates that no value was available. The second and third rules correspond, respectively, to the non empty and empty input channel patterns. A diamond with code 2 indicates the end of a channel (or of a list value). In this case, being activated or not (a variable in the preceding diamond) the process generates the last value by putting its parameter in the output channel, and then disappears.

As said before, nets always have two rules (interface and expansion). The net behaviour depends on the internal structure of interconnections among its processes, given by the expansion rule. The interface rule allows that a process or a net (provided it has a single output channel) can act as a process parameter inside other nets. In this context, supporting full higher order implies the dynamic
transformation of the net in such a way that a functional parameter (a function name as + in Figure 9.5) can act, i.e. be converted in an executable subnet (as the resulting third value in Figure 9.6). On the other hand partially applied functions (processes having big arrows in some of their entries) can also be a valid parameter or result, and therefore partial parameter passing is also allowed. In EGG a set of only twelve rules handles higher order and partial instantiation in all possible cases. In the presence of processes acting as parameters the application of these rules causes ‘internal states’ to show up. These intermediate states of the net can be optionally hidden from the user.

Moreover, also functional values in NiMo are first class in the ‘showing’ sense, allowing incremental development and modular testing of subnets the final results of which contain functional values. For instance, once the net corresponding to the expression \( \text{map}(\text{zipWith}(\text{const id}))[1, [2], [3, 4]] \) is completely evaluated (no 0-value diamonds remains), it results into a channel of partially instantiated functions representing the expression \( \text{[zipwith(const id) [1], zipwith(const id) [2], zipwith(const id) [3, 4]].} \) On the contrary, in textual languages the expression is considered erroneous, as in Haskell, or else in Miranda the result shown is: \([\text{<function>, <function>, <function>}.\]

\[\text{FIGURE 9.11. Prefixes of the list [1,2,3]}\]

Regarding data, normal order evaluation leads to duplicate sub-expressions. In lazy functional languages identical substructures are evaluated only once using graphs instead of trees as syntax internal representation; several nodes share the same son. Data management policy in NiMo uses the same strategy and is implemented by sharing structured values. That means, that evaluated subnets can not be immediately discarded because they could be shared by another subnet. A set of garbage collector rules in EGG takes care of erasing ‘right to left’ disconnected (as provider) nodes. This set of rules act ‘in parallel’ with the rest of the grammar rules. Figure 9.11 displays an intermediate state of the net \( \text{prefixes} \) applied to a finite input channel \([1,2,3]\), exhibiting sub graph sharing.

In EGG, all the previously mentioned tasks are organized into sets of specific purpose rules. Besides, another advantage of having a uniform framework is that program transformation and optimization can be addressed using the same mechanism. A set of rules can do optimization using known patterns. This type of transformations could take place before the execution starts, or even when the
pattern shows up due to the net evolution.

### 9.4 CURRENT STATE AND FUTURE WORK

A running prototype has been implemented using AGG [ERT99], [TB04] as the graph transformation system. Currently, over 20 basic processes including equivalent versions for most useful functions in Haskell prelude are implemented. Real and tuple types are not yet included. A special effort was done in order to reduce the number of rules, keeping always in mind that the system time complexity strongly depends on the grammar size and on the sub-graph matching algorithm being used. The presence of diamonds, gives bipartite sub-graphs, and then heavily reduces the number of rules (patterns in left hand sides). Adding them all up, the sets of rules in EGG are around a hundred. On the other hand, the kernel grammar does not have critical pairs, and only injective match mappings are used, reducing the execution (graph transformation) complexity. Confluence (and therefore rule application order independence) assures a well-defined semantics.

The system has been tested on classical examples as hamming numbers, prime numbers in both static and dynamic versions, prefixes, quicksort in different dynamic and static versions, obtained by derivation from the classical one (Figure 9.7), other sorting algorithms, and of course, Fibonacci.

In order to test full higher order and partial application a set of examples were evaluated, for instance the NiMo nets equivalent to the following Haskell codes:

```haskell
zipWith id(map(zipWith(const id)) [[1],[2],[3,4]]) l
where l = [[5], [6,7,8], [9], []]
```

```haskell
zipWith id(zipWith const(zipWith id(repeat (+))nat)nat)nat
where nat = [1..]
```

The work done shows the feasibility of the approach and the experimental possibilities it opens. However, although the main design and implementation challenges have been overcome, for the time being, the current state of the system is far from being a final product. One of the main drawbacks of the current version is, precisely, the graphical interface. To be a really useful tool this aspect should be heavily improved. As the net grows, it is necessary to have control on scaling, zooming, scrolling and other facilities allowing 'pretty printing' of the net, not present in the graphic interface of the current transformation system. Nevertheless, the graph transformation engine can be used isolated and be combined with another interface. In fact, graph transformation systems are powerful tools to deal with complex graph configurations but, as far as the authors know, they are not intended to be used as graphical interfaces in a relatively large size application. They are mainly used for modelling. In NiM05 executions of some of the examples tested in the prototype are shown as they would look like with an improved graphic interface.

Regarding the extension of the language itself, as we said before, a more powerful type inference system is under development. And, in general, other Haskell features not having equivalence in NiMo, should also be considered. In particular, user defined types and therefore data abstractions mechanisms, which are a key
issue for scalability \[BBB^{+95}\]. On this regard, NiMo already has a procedural abstraction mechanism to represent a complex net by a single node, therefore allowing to face more complex problems. But, as in every visual language, as the net grows the limited screen size is a problem that the usual mechanisms (scrolling, zooming, etc.) are not powerful enough to solve. Currently, once expanded a net process remains so whether active or not. A possible approach for a better use of the screen real estate would be to provide operations to reverse the expansion, compress basic type-values, and display a graphic shorthand for numeric lists.

Finally as it can be seen in all the previous examples, translation from NiMo to Haskell is almost direct. NiMo is specially suited for debugging and experimentation, not for being the final execution system. The last step of the workbench should be to produce an equivalent but efficiently executable textual version, ideally in a parallel language.

9.5 CONCLUDING REMARKS

This paper has presented a novel graphical language inspired in the data flow representations for lazy functional programs, and the environment designed to give support to this programming style. The main characteristics of the language have been illustrated with several examples, showing how the program evolves in execution, and the particular kind of interactions the system allows. We have also described the fundamental ideas in the implementation of the environment by means of a graph transformation system, the current state of the implementation, its drawbacks and possible improvements.

The most relevant characteristics of NiMo are:

- It is totally graphic. Programs are graphs drawn by the user, which are directly executable without any kind of translation and evolve showing all transformations of data and processes, allowing online tracing. On the other hand, the environment has been fully implemented using graph grammars, and therefore the implementation of the environment itself is also graphic.

- The language is very simple and expressive, close to the usual data flow representations for lazy functional programs. Nets of growing complexity are built with a small set of graphic primitives. Only seven symbols are enough to represent and handle higher order, partial instantiation, polymorphism, type inference, and keep track of the evaluation state.

- The system provides a repertory of basic processes including equivalent versions of most useful functions in the Haskell prelude, but unlike functional programming, processes can have more than one output channel. This facility avoids the use of tuples and projections or symmetric duplication of processes making clearer the data flow solution.

- And the most innovative one is that the user has direct control on the computation state and can change data, processes and/or control at any point of the execution. This is feasible because of the combination of several peculiarities:
the source code is exactly the computation initial state, the activation or evaluation state is explicit, the execution can be done step-by-step, and between steps the user can modify the net. Therefore, not only online debugging is natural but even, laziness can be locally relaxed in an interactive way.

REFERENCES


