Embedded Compilers

Arthur Iwan Baars

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN: 978-90-8891-131-6
Author: Arthur Iwan Baars
Printed by: Boxpress, Oisterwijk
Cover: © Ciudad de las Artes y las Ciencias, Valencia, Spain, 2009

Embedded Compilers

Ingebedde Vertalers

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 9 december 2009 des ochtends te 10.30 uur

door

Arthur Iwan Baars

geboren op 1 november 1977 te Gouda

Promotor: Prof.dr. S. D. Swierstra

Abstract

For automation it is important to express the knowledge of the experts in a form that is understood by a computer. Each area of knowledge has its own terminology and ways of formulating things; be it by drawing diagrams, using formulae, or using formalized languages. In the last case we say we have a "Domain Specific Language", and –since it is formalised– it can usually be processed or analysed by a computer or even be compiled into machine code.

Domain-specific languages often do not have a formal specification and are usually designed and implemented in an ad-hoc fashion, frequently leading to inconsistent designs. Furthermore they often lack features that are usually found in programming languages, such as abstraction mechanisms, type systems, and static analysis. Such features are often considered to take too much effort to implement. Initially these features are not really missed, however, when programs grow, they become indispensable.

Defining a complete and consistent language from scratch is not an easy task, and so the question arises: how to support the design and implementation of domain-specific languages? A solution is to embed a domain-specific language in a general-purpose host language. This approach has many advantages. One does not have to implement an entirely new compiler, one can simply reuse the features of the host language, such as the type system and abstraction mechanisms. Futhermore, different embedded languages can be combined together in a single program.

In the Haskell community embedding domain-specific languages by means of combinator libraries is common practice. The rich type system and flexible notational features, such as user-defined operators, type classes, and **do**-notation, make Haskell very suitable for embedding domain-specific languages.

Originally combinator-based embedded languages directly expressed the denotational semantics of the embedded language. As a consequence, techniques used in conventional compilers are not applicable because the representation of the embedded progam is implicit. A logical next step along this line of development is to first build an intermediate structure, which can then be analyzed, transformed and optimized as in an ordinary compiler. How to do such things effectively in a strongly typed host language is the subject of this thesis: *Embedded Compilers*.

This thesis consists of the following papers:

- Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

- Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

- Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

Contents

Chapter 1

Introduction

1.1 Introduction

Since automation is becoming more and more important, the need for experts to be able to express their expertise, such that it is effectively understood by computers, increases as well.

Each area of knowledge has its own terminology, and ways of formulating things; be it by drawing diagrams, using formulae, or using formalized languages. In the last case we say we have a "Domain Specific Language", and –since it is formalised– it can usually be processed or analysed by a computer or even be compiled into machine code.

Domain-specific languages (DSL) often do not have a formal specification and are usually designed and implemented in an ad-hoc fashion, frequently leading to inconsistent designs. Furthermore they often lack features that are usually found in programming languages, such as abstraction mechanisms, type systems, and static analysis. Such features are often considered to take too much effort to implement. Initially these features are not really missed, however, when programs grow, they become indispensable.

Defining a complete and consistent language from scratch is not an easy task, and so the question arises: how to support the design and implementation of domain-specific languages? A solution is to embed a domain-specific language in a general-purpose host language. An embedded domain-specific language (EDSL) has many advantages over a normal domain-specific language (DSL). First of all the design and implementation of an EDSL is easier. One does not have to implement an entirely new compiler, but can make use of the host language's compilers and tools. Moreover a programmer can use the domain-specific notation, and at the same time benefit from all the features, such as the type system and abstraction mechanisms, of the general purpose host language.

A powerful type system, and many notational features, such as monad-comprehensions (**do**-notation), user-defined operators and type classes make a modern functional language such as Haskell an excellent tool for embedding domain

specific languages. Examples of EDSL's include HaskellDB [39] for database programming, QuickCheck[14] for software testing, Wash/CGI[61] for server-side web applications, Haskore for music composition [28], Yampa for reactive programming [18], and pretty printing [13, 29, 66, 48] and parsing-combinator [20, 64, 33, 59, 38, 31] libraries.

The usual approach for defining an EDSL is by defining a library of combinators, each combinator representing a grammatical structure from the embedded language. Because EDSL programs are in fact programs in the host language we can easily combine different EDSL's in a single program.

Originally combinator-based embedded languages directly expressed the denotational semantics of the embedded language. Techniques used in conventional compilers, however, are not directly applicable because the representation of the embedded progam is implicit. A logical next step along this line of development is to first build an intermediate structure, which can then be analyzed, transformed and optimized as in an ordinary compiler. How to do such things effectively in a strongly typed host language is the subject of this thesis: *Embedded Compilers*.

This thesis consists of five articles, which were originally published as:

- Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

- Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

- Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

Small adaptations were made to the original articles, to improve the consistency througout this thesis. The Haskell keyword `forall` is formatted as $\forall$ whenever it is used to denote a universally quantified type, and as $\exists$ when used to denote an existentially quantified type. Furthermore, the use of the term "Embedded Domain-Specific Language" and its acronym "EDSL" has been made consistent.

The first article presents support for "permutation parsers" as an extension to a parsing-combinator library, and is an example of how we can combine parsers in new and unexpected ways. It serves as a demonstration of how we can enforce certain properties of our program by means of a dynamically constructed parser, thus relieving the programmer from the cumbersome task of checking the program for these properties. The other four papers form a series and deal with the *Embedded Compilers* theme outlined above.

1.2 Parsing Permutation Phrases

The first paper was published as:

> Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

This paper extends a parsing-combinator library with support for permutation phrases. A parsing-combinator library is an embedded domain-specific language for creating parsers, and allows parsers to be expressed in a concise and natural notation that closely resembles that of EBNF grammars. At the same time a user has the full abstracting power of the underlying programming language at hand. Complex, oft-recurring patterns can be expressed by defining new combinators.

A specific parsing problem is the recognition of a permutation phrase. A permutation phrase is a sequence of elements in which each element occurs exactly once and for which the order is irrelevant. Applications include the generation of parsers for attributes of XML tags, Haskell's record syntax, and fields of BibTeXentries.

Cameron [9] proposed the following notation for permutation phrases as an extension of EBNF:

$$S ::= \langle\langle A \mid\mid B \mid\mid C \rangle\rangle$$

meaning that non-terminal S derives a sequence of A, B, and C in any order.

Our implementation in the form of a combinator library closely resembles this notation. The example above is written as:

$$s = \mathit{permute}\ ((,,) \mathbin{<\$>} a \mathbin{<||>} b \mathbin{<||>} c)$$

The call of the function *permute* (...), and the $<||>$ operators correspond, respectively, to the outer brackets and the || symbols. The $<\$>$ operator attaches a semantic action that combines the parsing results for the constituents of the permutation phrase. In this case the results of a, b and c are simply combined to form a triple.

A problem arises here: the constituents of the permutation phrase are to be recognized in any order; the semantic action, however, is a function and expects its arguments in a specific order. The implementation must automatically reorder the recognized elements to the order expected by the semantic action.

To complicate matters even more the constituent parsers may have results of different types.

To solve this problem the implementation uses a permutation tree as intermediate data structure. The data type for permutation trees uses existentially quantified types to deal with the fact that the results of the constituent parsers could have different types. Recognizing a permutation means following a path through the tree from the root to a leaf; each possible permutation corresponding to a specific path. In our implementation the leaf nodes at the end of each path contain a function that "knows" in which order the elements were recognized. It uses this knowledge to reorder the recognized components in the order in which they were specified in the code for the parser, and as a consequence the given semantic function can be used to combine the individual results into a single result for the whole parser. A permutation tree can be very large, since it contains a specific path for each possible permutation. Luckily lazy evaluation ensures that the algorithm constructs only those parts of the tree that are really used for recognizing a concrete input.

The library also supports optional elements and separator symbols. Implementations are available in both the monadic parsing library Parsec[38], and the error-repairing parsing-combinator library [59] developed at Utrecht University, and is nowadays also part of the *Control.Applicative.Permutation* Haskell module [43].

1.3 Typing Dynamic Typing

The second paper was published as:

> Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

This paper presents a type-safe implementation of dynamic typing for the functional programming language Haskell. Haskell is a statically typed programming language. This means that typing rules are checked at compile time. Type violations are reported before program execution, and efficient object code can be generated since no type consistency check needs to be performed at run time. However, in some situations the type of a value cannot be determined before run time. For example when a program interprets a (meta)language term, resulting in an object-language value of a type that depends on the specific term at hand. An example of such an interpretation is the function *eval* which takes a *String*, parses it into an expression and returns the value of that expression.

There exist many proposals [1, 2, 40] for dealing with dynamically typed values in a statically typed language. These are all based on a similar idea: extend the language with a universal type *Dynamic*, and embed dynamic values in that type. This new type is actually a pair of the value itself together with a representation of its type. Before using the dynamic value, its type component

is inspected using a *typecase* or similar construct. The Haskell language, however, does not have built-in support for dynamic typing. Instead, the Haskell base libraries contain a module *Data.Dynamic* that implements dynamic values. It provides a data type *Dynamic* for dynamically typed values. Furthermore, it provides a function *toDyn* that injects a value into a dynamically typed value, and a function *fromDyn* that converts a dynamic value into a concrete type. Although this implementation cannot deal with polymorphic values, it has proven to be very useful in practice. The implementation of this module is based on *unsafeCoerce*, and as the name suggests this function is not type-safe.

In this paper we develop a statically typeable library which provides a form of dynamic typing that is as powerful as Haskell's dynamic typing library, but does not make use of unsafe functions. The interface of the library is basically as follows:

$$\textbf{data}\ \mathit{Dynamic} = \exists a\ .\ \mathit{Dyn}\ a\ (\mathit{TypeRep}\ a)$$

The data type *Dynamic* is a pair of a value of type a and a representation (*TypeRep*) of the type a. Existential quantification is used to hide the type a.

The function *toDynamic* injects a value into a *Dynamic* value:

$$\begin{aligned}&\mathit{toDynamic}\ ::\ \mathit{TypeRep}\ a \to a \to \mathit{Dynamic}\\&\mathit{toDynamic}\ \mathit{tp}\ \mathit{val} = \mathit{Dyn}\ \mathit{val}\ \mathit{tp}\end{aligned}$$

The function *fromDyn* can be used to cast a *Dynamic* object into a value of a concrete type. The representation of the expected type is passed to *fromDyn*, which checks if the expected type matches the actual type of the dynamic value, and returns the value if the check succeeds. Because this check can fail the function returns a *Maybe* value.

$$\begin{aligned}&\mathit{fromDyn}\ ::\ \mathit{TypeRep}\ a \to \mathit{Dynamic} \to \mathit{Maybe}\ a\\&\mathit{fromDyn}\ \mathit{expected}\ (\mathit{Dyn}\ \mathit{val}\ \mathit{actual}) = \textbf{if}\ \mathit{actual} \sim \mathit{expected}\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{then}\ \mathit{Just}\ \mathit{val}\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ \ \mathit{Nothing}\end{aligned}$$

The implementation of *fromDyn* above is not accepted by Haskell's type checker. It does not "know" that the type of *val* is a when the expected and actual types match. Instead it complains that the type of *val* is existential and is not supposed to escape. In order to convince the type checker, we let the $(\sim)$ operator return a value of type *Equal* which encodes a proof that the actual and expected value coincide, and hence that it is safe to convert the actual value *val* to a value of the expected type a.

The data type *Equal* is the essential ingredient of the paper, and is inspired by the "Identity of indiscernibles" principle. This principle is attributed to Gottfried Leibniz (1646 - 1716) as is therefore also known as Leibniz' Law. Leibniz' Law states that a and b are equal if and only if all their properties are the same, and formally reads:

$$a \equiv b = \forall p\ .\ p\ a \leftrightarrow p\ b$$

This formula can be proven equivalent to:

$$a \equiv b = \forall p \ . \ p \ a \to p \ b$$

Encoded as a type according to the Curry-Howard isomorphism we get the following definition for the type *Equal*:

$$\textbf{data} \ \mathit{Equal} \ a \ b = \mathit{Equal} \ (\forall p \ . \ p \ a \to p \ b)$$

The identity function (*id*) is the only non-diverging function whose type is an instance of the type $(\forall p \,.\, p \ a \to p \ b)$. The existence of a value of type *Equal a b*, i.e. matching a constructor *Equal*, now implies that $a \equiv b$, since the conversion function inside must be the identity function. Hence we can define a function *coerce* that, given a proof that the types *a* and *b* are equal, safely converts a value of type *a* into type *b*:

$$\begin{array}{l}\textbf{data} \ \mathit{Id} \ a = \mathit{Id} \ a \\ \mathit{coerce} :: \mathit{Equal} \ a \ b \to a \to b \\ \mathit{coerce} \ (\mathit{Equal} \ \mathit{conv}) \ a = \textbf{case} \ \mathit{conv} \ (\mathit{Id} \ a) \ \textbf{of} \ \mathit{Id} \ b \to b\end{array}$$

In the paper the operator $(\sim)$ that matches two type representations is given the following type:

$$(\sim) :: \mathit{TypeRep} \ a \to \mathit{TypeRep} \ b \to \mathit{Maybe} \ (\mathit{Equal} \ a \ b)$$

This function checks whether the two type representations are equivalent, and if so, returns a proof that their two corresponding types are equal. Using the operator $(\sim)$ and *coerce* we can define the function *fromDyn* as follows:

$$\begin{array}{l}\mathit{fromDyn} :: \mathit{TypeRep} \ a \to \mathit{Dynamic} \to \mathit{Maybe} \ a \\ \mathit{fromDyn} \ \mathit{expected} \ (\mathit{Dyn} \ \mathit{val} \ \mathit{actual}) = \textbf{case} \ \mathit{actual} \sim \mathit{expected} \ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just} \ \mathit{eq} \to \ \mathit{Just} \ (\mathit{coerce} \ \mathit{eq} \ \mathit{val}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}\end{array}$$

The existential type of *val* is now coerced to the expected type *a* before returning. This definition is accepted by the type checker, since the existential type is not escaping.

The paper presents an implementation of the data type *TypeRep* and a small set of proof combinators that construct values of type *Equal* in a compositional way. As an example of the use of dynamic typing the paper shows a small interpreter for a simple expression language. It takes as input an untyped abstract syntax term, interprets it, and returns the result as a dynamic value.

Conclusions

The type *Equal* has proven very useful for implementing meta-programming with typed abstract syntax [45]. Consider for example the abstract syntax of a very simple expression language containing *Bool* and *Int* constants and an **if** statement:

```
data Expr a = IntVal Int      (Equal Int a)
            | BoolVal Bool (Equal Bool a)
            | If (Expr Bool) (Expr a) (Expr a)
            | ...
```

An evaluator for this language can be defined as follows:

```
eval :: Expr a → a
eval (IntVal i eq)  = coerce eq i
eval (BoolVal b eq) = coerce eq b
eval (If c t e)     = if eval c then eval t else eval e
```

A great advantage of typed abstract syntax is that one can only construct object-language terms that respect the object-language's typing rules, making it very suitable for implementing domain-specific embedded languages.

The type *Equal* inspired several works on typed abstract syntax and phantom types, which are, among others, the motivation for extending GHC with Generalized Algebraic Data types (GADT). Programming using GADTs is quite similar to programming using the type *Equal*. In fact both are about as expressive. The great benefit of GADTs is that all the tedious construction of equality proofs is done by the compiler. For this GHC's typed intermediate language has been extended with "equality coercions" [57]. This makes programming much more pleasant, and furthermore, since the equality coercions can be erased, no run time penalty is paid for their evaluation.

1.4 Type-safe Self-inspecting code

The third paper was published as:

> Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

This paper focuses on a representation of typed abstract syntax trees in with observable recursive binding structures. The paper uses a parsing-combinator library as an example of a domain-specific embedded language. In fact the paper implements an embedded compiler compiler; work that is normally performed by a compiler compiler such as YACC [35] is done instead in a typed setting, in the embedded language's implementation.

Parsing-combinators are a well-known example of an embedded domain-specific language. It is implemented as a library of combinators in such a way that parsers can be defined in a notation that closely resembles that of EBNF. However, one runs into problems, when straightforwardly translating a grammar in a combinator-based parser. If the grammar is left-recursive then the resulting parsers will not terminate due to the top-down parsing strategy being

employed. When several alternative productions share common prefixes the resulting parser might be unexpectedly inefficient. Such deficits can be fixed by manually applying left-recursion removal and left-factorization transformations. However, for larger grammars applying those transformation by hand is tedious and often results in messy code, effectively loosing notational elegance, which is supposed to be the greatest selling point for domain-specific languages.

A solution would be to let the combinators apply the transformations themselves, so we maintain notational elegance and *at the same time* obtain efficient parsers. To achieve this we need an intermediate representation of the parsers. This intermediate representation must be analyzed and transformed and finally "compiled" into a real parsing function. The intermediate representation should allow us to detect cycles. Furthermore we need to ensure that the transformations preserve the well-typedness of the parsers.

The key ingredient of the paper is an implementation of typed abstract syntax. The typed abstract syntax for a combinator parser is defined as follows:

$$\begin{aligned}\textbf{data}\ \mathit{Parser}\ \mathit{env}\ a = {} & \mathit{Symbol}\ (\mathit{Equal}\ \mathit{Char}\ a)\ \mathit{Char}\\ \mid {} & \mathit{Succeed}\ a\\ \mid {} & \mathit{Fail}\\ \mid {} & \mathit{Choice}\ (\mathit{Parser}\ \mathit{env}\ a)\ (\mathit{Parser}\ \mathit{env}\ a)\\ \mid {} & \exists b\ .\ \mathit{Seq}\ (\mathit{Parser}\ \mathit{env}\ (b \to a))\ (\mathit{Parser}\ \mathit{env}\ a)\\ \mid {} & \mathit{NT}\ (\mathit{Ref}\ \mathit{env}\ a)\end{aligned}$$

The abstract syntax has an alternative for each of the basic combinators. The *Equal* data type of the previous paper is used to ensure that in the case of a *Symbol* constructor the associated type a is equal to *Char*. The last alternative (NT) represents an occurrence of a non-terminal and contains a reference to another production (*Parser*). Such references have type *Ref* and are labelled with a type a representing the type of the *Parser* to which the reference refers. Furthermore, *Ref*s and *Parser*s are labelled with a type variable *env*. The type *env* represents the type of the grammar. A grammar is a collection of productions (*Parser*s). The purpose of the *env* type variable is to ensure that the references used in a production are guaranteed to refer to productions that are contained in the grammar, and furthermore recognize a value of a consistent type. As a result Haskell's type system ensures that we cannot have dangling references or incorrectly typed semantic actions.

The type *Ref* represents references and is defined as follows:

$$\begin{aligned}\textbf{data}\ \mathit{Ref}\ \mathit{env}\ a = {} & \exists \mathit{env}'\ \ .\ \mathit{Zero}\ (\mathit{Equal}\ \mathit{env}\ (a, \mathit{env}'))\\ \mid {} & \exists \mathit{env}'\ x\ .\ \mathit{Suc}\ \ (\mathit{Equal}\ \mathit{env}\ (x, \mathit{env}'))\ (\mathit{Ref}\ \mathit{env}'\ a)\end{aligned}$$

The type *Ref env a* represents a reference to an object of type a that is located somewhere in an environment of type *env*. The type variable *env* is a nested cartesian product representing the types of the objects in the environment. A *Ref* can be seen as an index into this nested cartesian product. It is implemented as a Peano numeral. The constructor *Zero* is a reference that refers to the first object in *env*, hence the *env* must be of the form $(a, \_)$, as stated in the

associated *Equal* type. In case of a *Suc* constructor the reference does not refer to the first object but to an object somewhere further down in the environment.

The most important operations on references are the following:

$$\begin{aligned} &\mathit{equalRef} :: \mathit{Ref}\ \mathit{env}\ a \to \mathit{Ref}\ \mathit{env}\ b \qquad \to \mathit{Maybe}\ (\mathit{Equal}\ a\ b) \\ &\mathit{deref} \quad :: \mathit{Ref}\ \mathit{env}\ a \to \mathit{Env}\ f\ \mathit{env}\ \mathit{env} \to f\ \mathit{env}\ a \end{aligned}$$

The function *equalRef* allows us to compare references for equality, and is essential for detecting cycles. If the two *Ref*s are indeed equal a proof is returned that states that the objects to which they point have the same type. The function *deref* is used to look up the object corresponding to a *Ref*. The type variable f, of kind $* \to * \to *$, stands for the type of the objects in the environment. In the case of parser combinators: $f \equiv \mathit{Parser}$.

Conclusions

The paper contains an implementation of parsing-combinators that analyze themselves and automatically remove left-recursion. Instead of building a parsing function directly, the combinators first build a typed abstract syntax containing explicit references, which is subsequently transformed. After the transformation the abstract syntax is "compiled" into a parsing function.

Though the paper focusses on parsing-combinators, the techniques employed can be used in a much wider setting, since it enables the inspection and transformation of any program structure that contains binding structures. The use of typed abstract syntax ensures the type correctness of the transformed program structures.

1.5 Typed Transformations of Typed Abstract Syntax

The fourth paper was published as:

> Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

run time transformations of embedded languages can have a huge effect on performance. In earlier work Viera et al. [63] presented a case study comparing the *Read* instances generated by Haskell's **deriving** construct with instances on which run time grammar transformations (precedence resolution, left-factorization, and left-recursion removal) have been applied.

In this paper we present the library underlying the implementation of the transformations used in that case study, and demonstrate its use in implementing common sub-expression elimination. The library uses typed abstract syntax as defined in the previous paper to represent fragments of embedded programs

containing variables and binding structures, while preserving the idea that the type system of the host language is used to emulate the type system of the embedded language. The tricky issue is how to to keep a collection of mutually recursive structures well-typed while being transformed.

A group of declarations of the program being transformed is represented as a heterogenous collection of abstract syntax terms. The type of such a collection is defined using a GADT as follows:

```
data Env term use def where
  Empty :: Env term use ()
  Ext   :: term use a → Env term use def' → Env term use (a, def')
```

In this definition the type variable *term* stands for the type of the terms being transformed. For example, in case of common sub-expression elimination those terms are expressions, and for grammar transformations the terms represent productions. The type variable *def* is a nested cartesian product containing the types of the declarations contained in the *Env*. The variable *use* on the other hand records which declarations are used by the terms in the *Env*. When *use* and *def* coincide then the group of declarations is closed, i.e. each declaration used in the declaration group is also defined in that group.

During a transformation a group of declarations does not need to be closed. It may very well be that terms contain references that refer to declarations that are to be added by later transformation steps. At the end of the transformation, however, the declaration group must be closed. In this way the type checker verifies that all references point to values of the right types and that there are no dangling references. References in the typed abstract syntax terms are represented by the type *Ref*, as explained in the previous paper. Basically a *Ref* is the index (encoded as Peano number) of the declaration to which the reference points. Whenever a transformation step inserts new declarations the existing indexes become invalid, and should be "incremented" with the number of new declarations. To avoid having to do this tedious work manually, the transformation library automatically ensures that all existing indexes remain consistent whenever new declarations are added by a transformation step.

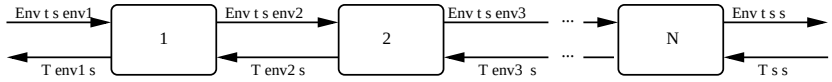

Figure 1.1:

The way this problem is solved is graphically depicted in Figure 1.1. In this figure each block represents a step in the transformation process. During the transformations a collection of declarations is being constructed (*Env*), and each step may insert new declarations. Recall the arguments of the type *Env term use def*; where *term* is the type of the terms being transformed, *use* is the type of the references used in those terms, and *def* contains the types of the declarations in the *Env* block.

The problem was that in each transformation step the references need to be incremented depending on the number of declarations added by future steps. The required information about the "future" is passed backwards through the computation (from right to left in the figure) in the form of a *Ref*-transformer. A *Ref*-transformer effectively increments references, and has the following type:

$$\textbf{data}\ T\ from\ to = T\ (\forall x\ .\ Ref\ x\ from \rightarrow Ref\ x\ to)$$

The collection of transformed declarations, on the other hand, is computed left to right. Its type is of the form *Env t s def*, where *def* is a cartesian product consisting of the types of the declarations inserted thus far. In the figure the *def* component changes in each transformation step (as a result of new declarations being inserted), until we reach the final transformation step, where the *def* component is equal to *s*.

Conclusions

The basis of our transformation library is a data type *Trafo* representing transformation steps. Its implementation is based on the idea presented in Figure 1.1, augmented with *Arrow* in- and output and extra state, to communicate values between transformation steps. Furthermore the data type *Trafo* is made an instance of the *Arrow* class, allowing transformation steps to be combined in a compositional way using the special *Arrow* notation.

Unfortunately, our initial design is not accepted by the GHC compiler, because it makes use of lazy pattern matching on an existentially quantified datatype. This feature is not supported by GHC because it cannot be translated to the intermediate core language of the compiler which is based on System-F. Other compilers such as Hugs [32] and UHC [62] do support lazy matching on data constructors with existentially quantified types. To overcome this problem we present two solutions. The first is a bit of a cheat, using *unsafeCoerce* to prevent GHC from complaining. The other solution uses a slightly changed version of the *Trafo* data type. Apart from a slightly changed type all code implementing the operators of the library is essentially the same. The downside is that one cannot use the special *Arrow* notation anymore. Due to the required changes to the *Trafo* type the alternative implementation of the library no longer implements a true *Arrow*. We hope that GHC will implement lazy matching on data constructors with existentially quantified types in the future. Until then we have to settle with a cheat or do without the *Arrow* notation when defining transformations on typed abstract syntax.

1.6 Typed Transformations of Typed Grammars The Left Corner Transform

The final paper was published as:

> Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

This paper presents an implementation of the left-corner transform, which removes left-recursion from a grammar. The third paper also presented an algorithm for removing left-recursion. This algorithm, however, results in the worst case in an exponential increase of the grammar size. The left-corner transform is much better and has a worst-case increase of only $O(n^2)$, where n is the number of symbols in the grammar. The left-corner transform introduces many, albeit small, new productions. The transformation library presented in the previous paper is used to maintain this growing set of productions.

The left-corner transform consists of repeatedly applying the following rules on the productions of a grammar:

Rule 1 For each production $A \rightarrow X\ \beta$ of the original grammar add $A\_X \rightarrow \beta$ to the transformed grammar, and add X to the left-corners of A.

Rule 2 For each newly found left-corner X of A:

- **a** If X is a terminal symbol add $A \rightarrow X\ A\_X$ to the new grammar.
- **b** If X is a non-terminal then for each original production $X \rightarrow X'\ \beta$ add the production $A\_X' \rightarrow \beta\ A\_X$ to the new grammar and add X' to the left-corners of A.

The paper firstly presents a simple implementation of the left-corner transform on an untyped representation of a grammar. An implementation for a typed grammar representation is subsequently derived from the untyped variant.

1.7 Conclusions

In this thesis techniques from conventional compiler construction are used in the implementation of embedded languages: the embedded program is first represented as a typed abstract syntax tree, which is subsequently analyzed and transformed, and finally "compiled" into a function representing its semantics. Hence the title: "Embedded Compilers".

An important aspect of embedding a language is that the embedded language inherits its type system from the host language. For this we have to ensure that the abstract syntax trees representing the embedded programs represent the types from the embedded language. It is remarkable that this can be done in Haskell itself using the technology from the second and third paper. This pattern has become so common that it has led to the introduction of generalized algebraic data types (GADTs) into Haskell.

The embedding approach has many advantages. The implementation effort for an embedded language is fairly small, because one does not need to develop

a compiler from scratch. The embedded language is implemented as a library, and embedded programs are actually normal programs in the host language. Because of this, various embedded languages can be readily combined in the same program, and moreover, the tools and development environments of the host language can be easily reused for the embedded languages.

The embedded approach has also disadvantages. Although a domain expert might think that he writes his program in a domain-specific language; in fact, he is still programming in the host language. As a result errors in his embedded program are reported as (type) error messages from the host language's compiler. This problem has been addressed in the thesis of Heeren [25, 24]. Heeren proposes the use of "type inferencing directives" to customize the behaviour of and error messages reported by the type checker. This approach has been successfully implemented in the Helium compiler [26].

Another disadvantage is that the notation that can be used in an embedded language is limited by the flexibility of the host language. A solution to this is to use a smart preprocessor such as Camlp4 [42] or syntax macros [37, 10, 4] as briefly discussed in the third paper. These approaches allow for nearly limitless freedom in the definition of new notation. This new notation is expanded to host language constructs. As a result error messages are presented in the expanded host language syntax, which makes it very difficult for a programmer to understand the error messages and locate the source of the problem. A solution for this, in a Haskell setting, is presented in the Master's Thesis of Rommes [53]. His approach is an extension of the idea of syntax macros. Instead of only defining new notation, a developer of an embedded language can also selectively redefine the attributes that make up the semantics of the language. This approach is very flexible and allows one to customize every part of a compiler to one's needs.

This concludes the introduction. The rest of the thesis consists of the papers introduced above.

Chapter 2

Parsing Permutation Phrases

This is a slightly edited version of a paper originally published as:
Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

FUNCTIONAL PEARL
Parsing Permutation Phrases

Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra
Institute of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands

Abstract: A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. We show how to extend a parser combinator library with support for parsing such free-order constructs. A user of the library can easily write parsers for permutation phrases and does not need to care about checking and reordering the recognized elements. Applications include the generation of parsers for attributes of XML tags and Haskell's record syntax.

2.1 Introduction

Parser combinator libraries have proved to be a very fruitful application area of functional programming languages: higher-order functions and the possibility to define new infix operators allow parsers to be expressed in a concise and natural notation that closely resembles the syntax of EBNF grammars. At the same time, the user has the full abstraction power of the underlying programming language at hand. Complex, often recurring patterns can be expressed by defining new combinators.

A specific parsing problem is the recognition of permutation phrases. A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. Since permutation phrases are not easily expressed by a context-free grammar, the usual approach is to tackle this problem in two steps: first parse a relaxed version of the grammar, then check whether the recognized elements form a permutation of the expected elements. This method, however, has a number of disadvantages. Dealing with a permutation of typed values is quite cumbersome, and the problem is often avoided by encoding the values in a universal representation, thus adding an extra level of interpretation. Furthermore, because of the two steps involved, error messages cannot be produced until a larger part of the input has been consumed, and special care has to be taken to make them point to the right position in the code.

Permutation phrases have been proposed by Cameron [9] as an extension to EBNF grammars, not aiming at greater expressive power, but at more clarity.

Cameron also presents a pseudo-code algorithm to parse permutation phrases with optional elements efficiently in an imperative setting. It fails, however, to address the types of the constituents.

We show how to extend any existing parser combinator library with support for parsing permutations of typed, potentially optional elements.

Possible applications include the implementation of Haskell's *read* function where it is desirable to parse the fields of a data type with labelled fields in any order, and the parsing of XML tags which have large sets of potentially optional attributes that may occur in any order. For instance, a parser for the XHTML image tag with some of its attributes can be written as follows:

```
imgtag          = token "<"  ⋗ token "img"  ⋗ attrs ⋖  token "/>"
  where attrs = permute (Img <$> field          "src"       uri
                             <|> field          "alt"       string
                             <|> optional (field "longdesc" uri)
                             <|> optional (field "height"   int)
                             <|> optional (field "width"    int)
                        )
```

The combinator <|> is used to separate parsers for permutable elements, and <$> can be used to apply a semantic function. Parsers for permutation phrases have to be enclosed by a call to *permute*. Our approach makes use of two features that are not provided by all functional programming languages: existentially quantified data types are used to encode reordering information that permutes the recognized elements to a canonical order. Additionally, we utilize lazy evaluation to make the resulting implementation efficient. The administrative part of parsing permutation phrases has a quadratic time complexity in the number of permutable elements. The size of the code, however, is linear in the number of permutable elements.

We therefore choose Haskell as implementation language. Existential types are not part of the Haskell 98 standard [46], but are supported by several current Haskell implementations.

The paper is organized as follows: Section 2.2 explains the parser combinators we build upon. Section 2.3 presents the idea of dealing with permutations in terms of permutation trees and explains how such trees are built and converted into parsers. In Section 2.4, we take a brief look at the applications mentioned above: the parsing of data types with labelled fields and the parsing of XML attribute sets. Section 2.5 concludes.

2.2 Parsing using combinator libraries

The use of a combinator library for describing parsers instead of writing them by hand or generating them from a separate formalism is a well-known technique in functional programming. As a result, there are several excellent libraries around. For this reason we just briefly present the interface we will assume in subsequent

```
infixl 3 <|>
infixl 4 <*>
class Parser p where
  fail    :: p a
  succeed :: a → p a
  symbol  :: Char → p Char
  (<*>)   :: p (a → b) → p a → p b
  (<|>)   :: p a → p a → p a
```

Figure 2.1: Type class for parser combinators

sections of this paper, but do not go into the details of the implementation. We want to stress, however, that our extension is not tied to any specific library.

We make use of a simple interface [52, 58] that is parametrized by the result type of the parsers and assumes a list of characters as input. It can easily be implemented by straightforward list-of-successes parsers [20, 64]. Our permutation parsers have also been implemented for more advanced libraries, such as the fast, error-correcting parser combinators of Swierstra [59] and the monadic-style [33] combinator library Parsec [38].

The parser interface used here is given as a type class declaration in Figure 2.1. The function *fail* represents the parser that always fails, whereas *succeed* never consumes any input and always returns the given result value. The parser *symbol* accepts solely the given character as input. If this character is encountered, *symbol* consumes and returns this character, otherwise it fails. The <*> operator denotes the sequential composition of two parsers, where the result of the first parser is applied to the result of the second. The operator <|> expresses a choice between two parsers. Finally, the application operator <$> is a parser transformer that can be used to apply a semantic function to a parse result. It can be defined in terms of *succeed* and <*>.

Many useful combinators can be built on top of these basic ones. A small selection that we use in this paper is presented in Figure 2.2.

2.3 Permutation parsers

We generate the parser for a permutation phrase from an intermediate tree that can be built using a set of special combinators. We first introduce an auxiliary data type to represent such permutation trees, and we show how to convert a permutation tree into a corresponding parser. We then investigate how permutation trees can be modified to be able to handle permutations containing optional elements. After that, we direct our attention toward how permutation trees can be built incrementally. In the final subsection, we demonstrate how to deal with separators in permutation phrases.

```
infixl 4 <$>, <$ , *>, <*
(<$>)      :: Parser p ⇒ (a → b) → p a → p b
f <$> p    = succeed f <*> p
(<$ )      :: Parser p ⇒ a → p b → p a
x <$ p     = const x <$> p
(<* )      :: Parser p ⇒ p a → p b → p a
p <* q     = const <$> p <*> q
( *>)      :: Parser p ⇒ p a → p b → p b
p *> q     = flip const <$> p <*> q
parens     :: Parser p ⇒ p a → p a
parens p   = symbol '(' *> p <* symbol ')'
```

Figure 2.2: Some useful parser combinators

2.3.1 Permutation trees

A permutation phrase of a set of elements can be expanded into an EBNF definition by summing up all possible permutations of the elements. Consider for example the permutation phrase of three elements a, b, and c. Using Cameron's notation for permutation phrases, we can write it as:

$$s ::= \langle\langle a \mid\mid b \mid\mid c \rangle\rangle$$

Expanding and subsequently left-factorizing this permutation phrase gives us the following EBNF production rule:

$$\begin{array}{lll} s & ::= & a\ b\ c \mid a\ c\ b \\ & \mid & b\ a\ c \mid b\ c\ a \\ & \mid & c\ a\ b \mid c\ b\ a \\ & \rightsquigarrow & \\ s & ::= & a\ (b\ c \mid c\ b) \\ & \mid & b\ (a\ c \mid c\ a) \\ & \mid & c\ (a\ b \mid b\ a) \end{array}$$

The left-factorized production rule can be represented by a tree as illustrated in Figure 2.3. Each path from the root to a leaf in the tree represents a particular permutation. Permutations with a common prefix share the same sub-tree, hence the number of choices in each node is limited by the number of permutable elements. Such a permutation tree is very suitable as intermediate data structure for a permutation parser. Parsing a permutation phrase boils down to checking whether the input matches one of the paths in the permutation tree.

If the grammar (and thus the permutation tree) is ambiguous, large parts of the tree might need to be evaluated before it can be decided which path must be followed. Therefore, ambiguous grammars should (as always) be avoided.

However, if the ambiguity in the grammar stems from optional elements in the permutation phrase, the permutation tree can be modified in a simple way to resolve the ambiguity.

In Section 2.3.2, we develop an implementation for permutations without optional elements, and in Section 2.3.3 extend this solution to cover optionality.

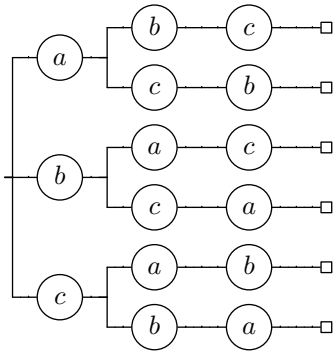

Figure 2.3: A permutation tree containing three elements

2.3.2 Permutation trees without optional elements

We introduce a data type *Perms* for permutation trees, which is parametrized by a type constructor p (e.g. the parser type) and a result type a.

$$\begin{aligned}&\textbf{data}\ \mathit{Perms}\ p\ a\ = \mathit{Empty}\ a\\ &\qquad\qquad\ \ \mid\ \mathit{Choice}\ [\mathit{Branch}\ p\ a]\\ &\textbf{data}\ \mathit{Branch}\ p\ a = \exists x\ .\ \mathit{Br}\ (p\ x)\ (\mathit{Perms}\ p\ (x \to a))\end{aligned}$$

The *Empty* constructor represents the leaves of the tree and the *Choice* constructor the branching nodes. A branch is constructed using *Br* and consists of an element, represented as a parser, plus a sub-tree. As the types of the elements may differ between branches, we hide their types by existentially quantifying the x in the definition of *Br*. The sub-trees have an element type that is different from the type of the original tree, making *Perms* a non-regular data type. Each sub-tree must contain information how to construct a value of the result type a from a value of the element's type. This is achieved by storing in the leaf at the end of each path a function that effectively reorders the elements on the path to construct a value of the result type.

To show that reordering is determined by the type of the components, we will henceforth write the type (or the element type for type constructors) of a variable as an index to its name.

The idea that each path in the tree represents the parser for one of the possible permutations is reflected by the following simple conversion function from permutation trees to parsers.

$$
\begin{array}{ll}
\mathit{permute} & :: \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \to p\ a \\
\mathit{permute}\ (\mathit{Empty}\ v_a) & = \mathit{succeed}\ v_a \\
\mathit{permute}\ (\mathit{Choice}\ bs_a) & = \mathit{foldr}\ (<|>)\ \mathit{fail}\ (\mathit{map}\ \mathit{pars}\ bs_a) \\
\quad \textbf{where}\ \mathit{pars}\ (\mathit{Br}\ p_x\ t_{x\to a}) & = \mathit{flip}\ (\$) <\$> p_x <\!\!*\!\!> \mathit{permute}\ t_{x\to a}
\end{array}
$$

Lazy evaluation plays an important role here, in that it ensures that the permutation tree is never computed completely. In fact, for a permutation of n elements, just the n tree elements immediately below the root of the tree are required to decide which sub-tree will be used to parse the rest of the permutation. From then on, only that sub-tree (a permutation tree of size $n-1$) is relevant. Iterating this argument leads to a complexity of only $O(n^2)$.

There are two potential problems here. First, the underlying parser combinator library might try to optimize parsing behaviour by evaluating different possible paths. This is problematic because permutation trees are so large that the precomputation is clearly undesirable. If the library has such features, they should be locally disabled for permutation trees. Second, repeated parsing of different permutations might cause multiple paths of the tree to be evaluated. In practice, however, the number of different permutations that actually occur in the input is small compared to the number of possible permutations.

2.3.3 Permutation trees with optional elements

Optional elements can be represented by parsers that can recognize the empty string and return a default value for this element. However, if *permute* is called on a permutation tree that contains such parsers, the resulting parser is ambiguous.

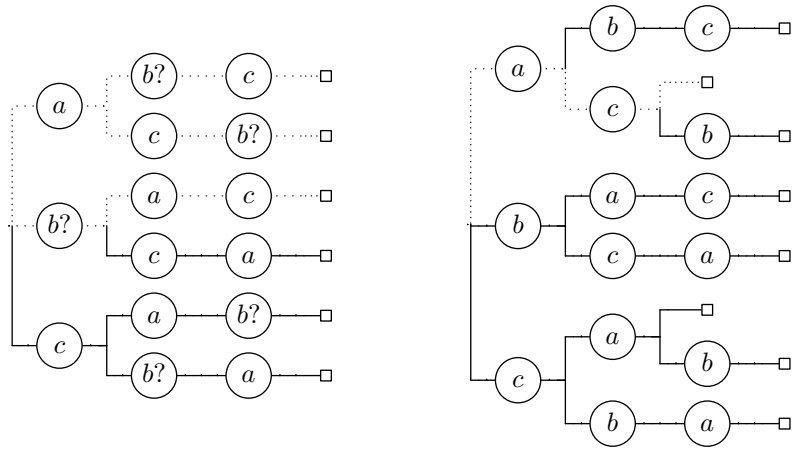

Figure 2.4: The ambiguous and the adapted permutation tree for optional b

Consider the left tree in Figure 2.4, which contains all permutations of a, an

optional b and c. Suppose we want to recognize ac. This can be done in three different ways since the empty b can be recognized before a, after a or after c. The three possible parses are shown as dotted paths in the figure. Fortunately, it is irrelevant for the result of a parse where exactly the empty b is derived, since order is not important. This allows us to use a strategy similar to the one proposed by Cameron [9]: parse nonempty constituents as they are seen and allow the parser to stop if all remaining elements are optional. When the parser stops the default values are returned for all optional elements that have not been recognized. The right tree in Figure 2.4 depicts this strategy for our three element example. The additional leaves mark the positions where the parser is allowed to stop. The string ac can now be parsed in only one way.

To implement this strategy we need to be able to determine whether a parser can derive the empty string and split it into its default value and its non-empty part, i.e. a parser that behaves the same except that it does not recognize the empty string. Both parts are represented as *Maybe* values: the default part is *Nothing* if and only if the parser cannot recognize the empty string; if the non-empty part is *Nothing*, the parser is either a *succeed* (i.e. it just carries semantics) or it is a *fail*. The splitting of parsers is represented by the *ParserSplit* class that is an extension of the normal *Parser* class. If the underlying parser combinator library cannot be easily adapted to cover this extension, one can alternatively introduce additional combinators, similar to $\langle|\rangle$ and $\langle\!\langle\$\rangle\!\rangle$, and let the user mark the optional elements explicitly [38].

```
class Parser p ⇒ ParserSplit p where
  split :: p a → (Maybe a, Maybe (p a))
```

In the solution that does not deal with optional elements a parser for a permutation follows a path from the root of a permutation tree to a leaf, i.e. an *Empty* node. In the presence of optional elements, however, a parser may stop in any node that stores only optional elements. We adapt the *Perms* data type to incorporate this additional information. If all elements stored in a tree are optional then their default values are stored in *defaults*, otherwise *defaults* is *Nothing*. The parser stored in each *Branch* is not allowed to derive the empty string. We can express the former *Empty* constructor as a function now.

```
data Perms p a = Choice { defaults :: (Maybe a)
                        , branches :: [Branch p a]
                        }
empty x        = Choice (Just x) []
```

The function *permute* is straightforwardly adapted to the generalized data type:

```
permute                    :: Parser p ⇒ Perms p a → p a
permute (Choice d_a bs_a)  = foldr (<|>) exit (map pars bs_a)
  where exit               = case d_a of
                               Just v_a  → succeed v_a
                               Nothing → fail
        pars (Br p_x t_x→a) = flip ($) <$> p_x <*> permute t_x→a
```

2.3.4 Building a permutation tree

Permutation trees are created by adding the elements of the permutation one by one to an initially empty tree. The function *add* takes a pair consisting of an optional default value and a parser that does not recognize the empty string, and adds it to an existing tree.

```
add :: (Maybe a, p a) → Perms p (a → b) → Perms p b
add (d_a, p_a) t_{a→b}@(Choice d_{a→b} bs_{a→b}) =
  let ins (Br p_x t_{x→a→b}) =
          Br p_x (add (d_a, p_a) (mapPerms flip t_{x→a→b}))
  in  Choice (d_{a→b} `ap` d_a) (Br p_a t_{a→b} : map ins bs_{a→b})
```

Having a default value means that the parser described by the permutation tree can accept the empty string. Surely, for the constructed tree, that is only possible if both the original tree has a default value and d_a is not *Nothing*. Then, the new default value can be built from the two using function application. The function *ap* does exactly this – it is function application lifted to *Maybe* types.

```
ap                  :: Maybe (a → b) → Maybe a → Maybe b
ap (Just f) (Just x) = Just (f x)
ap _        _        = Nothing
```

We can add a new element (d_a, p_a) to a permutation tree by inserting it in all possible positions to every permutation that is already in the tree. The function *add* explicitly constructs the tree that represents the permutation in which p_a is the top element. Additionally, for each branch of the original tree, the top element is left unchanged, and (d_a, p_a) is inserted everywhere (by a recursive call to *add*) in the sub-tree. Because the new element and the top element of the branch are now swapped, the function resulting from the sub-tree of the branch gets its arguments passed in the wrong order, which is repaired by applying *flip* to that function.

The function *mapPerms* is a mapping function on permutation trees. In a branch, $f_{a\to b}$ is composed with the function that results from the sub-tree.

```
mapPerms                    :: (a → b) → Perms p a → Perms p b
mapPerms f_{a→b} (Choice d bs) = Choice (fmap f_{a→b} d)
                                        (map (mapBranch f_{a→b}) bs)
mapBranch                   :: (a → b) → Branch p a → Branch p b
mapBranch f_{a→b} (Br p_x t_{x→a}) = Br p_x (mapPerms (f_{a→b}∘) t_{x→a})
```

We now define two operators for building permutation trees. The first is an operator that extends a permutation tree with a new element.

```
(<|>) ::      ParserSplit p ⇒ Perms p (a → b) → p a → Perms p b
perms <|> p = case split p of
  (Just e  , Just ne ) → add (Just e  , ne) perms   -- optional element
  (Nothing, Just ne ) → add (Nothing, ne) perms   -- required element
```

```
(Just e   , Nothing) → mapPerms ($e) perms      -- pure semantics
(Nothing, Nothing) → Choice Nothing []          -- fail
```

The second provides an empty permutation tree with initial semantics. It has a similar functionality as the <$> for normal parsers.

```
(<$$>)   :: ParserSplit p ⇒ (a → b) → p a → Perms p b
f <$$> p = empty f <|> p
```

An example with three permutable elements, corresponding to the tree in Figure 2.3, can now be realized by:

```
permute ((,,) <$$> int <|> char <|> bool)
```

where *int*, *char*, and *bool* are parsers for literals of type *Int*, *Char*, and *Bool*, respectively. Then all permutations of an integer, a character and a boolean are accepted, and the results of a successful parse are combined using the triple constructor (,,), thus yielding a value of type (*Int*, *Char*, *Bool*).

2.3.5 Separators

Permutable elements are often separated by symbols that do not carry meaning – typically commas or semicolons. Consider extending the three-element example to the Haskell tuple syntax: not just the elements, but also the parentheses and the commas should be parsed. Since there is one separation symbol less than there are permutable elements, our current variant of *permute* cannot handle this problem.

Therefore we define *permuteSep* as a generalization of *permute* that accepts an additional parser for the separator as an argument. The semantics of the separators are ignored for the result.

```
permuteSep              :: Parser p ⇒ p b → Perms p a → p a
permuteSep sep perm = permuteSep' (succeed ()) sep perm
```

The function *permuteSep′* now converts a permutation tree into a parser in almost the same way as the former *permute*, except that before each permutable element a separator is parsed. To prevent that a separator is expected before the first permutable element, we make use of the following simple trick. The *permuteSep′* function expects two extra arguments: the first one will be parsed immediately before the first element, and the second will be used subsequently. Using *succeed* () as first extra argument in *permuteSep* leads to the desired result.

```
permuteSep' :: Parser p ⇒ p c → p b → Perms p a → p a
permuteSep' fsep sep (Choice d_a bs_a) = foldr (<|>) exit (map pars bs_a)
  where exit = case d_a of
                 Just v_a  → succeed v_a
                 Nothing → fail
```

$$\begin{aligned} pars\ (Br\ p_x\ t_{x\to a}) = flip\ (\$) &\lhd\$\ fsep \\ &\lhd\!\rhd\ p_x \\ &\lhd\!\rhd\ permuteSep'\ sep\ sep\ t_{x\to a} \end{aligned}$$

The *permute* function can now be implemented in terms of *permuteSep*.

$$\begin{aligned} &permute :: Parser\ p \Rightarrow Perms\ p\ a \to p\ a \\ &permute = permuteSep\ (succeed\ ()) \end{aligned}$$

To return to the small example, triples of an integer, a character, and a boolean – in any order – are parsed by:

$$parens\ (permuteSep\ (symbol\ \texttt{','})\ ((,,)\ \lhd\$\rhd\ int\ \lhd\!\rhd\ char\ \lhd\!\rhd\ bool))$$

2.4 Applications

2.4.1 XML attributes

We now demonstrate the use of the permutation parsers by showing how to parse XML tags with attributes. For simplicity, we just consider one tag (the `img` tag of XHTML) and only deal with a subset of the attributes allowed. In a Haskell program, this tag might be represented by the following data type.

$$\begin{aligned} \textbf{data}\ XHTML = Img\ \{ &src\ \ \ :: URI \\ &, alt\ \ \ :: String \\ &, longdesc :: Maybe\ URI \\ &, height\ \ :: Maybe\ Int \\ &, width\ \ \ :: Maybe\ Int \\ &\} \\ \mid\ &\ldots \end{aligned}$$

Our variant of the `img` tag has five attributes of three different types. We use Haskell's record syntax to keep track of the names. The first two attributes are mandatory whereas the others are optional. We choose the *Maybe* variant of their types to reflect this optionality. Our parser should be able to parse the attributes in any order, where any of the optional arguments may be omitted. For the parsing process, we ignore whitespace and assume that there is a parser

$$token :: Parser\ p \Rightarrow String \to p\ String$$

that consumes just the given token and fails on any other input.

Using the *permute* combinator, writing the parser for the `img` tag is easy:

$$\begin{aligned} &imgtag \ \ :: ParserSplit\ p \Rightarrow p\ XHTML \\ &imgtag \ \ = token\ \texttt{"<"}\ \ \ast\!\rhd\ token\ \texttt{"img"}\ \ \ast\!\rhd\ attrs \lhd\!\ast\ \ token\ \texttt{"/>"} \\ &\quad \textbf{where}\ attrs = permute\ \$\ Img\ \lhd\!\$\!\rhd\ field\ \ \texttt{"src"}\ \ uri \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \lhd\!\rhd\ field\ \ \texttt{"alt"}\ \ string \end{aligned}$$

```
                                  <|> optional (field "longdesc" uri)
                                  <|> optional (field "height"   int)
                                  <|> optional (field "width"    int)
optional        :: Parser p => p a -> p (Maybe a)
optional p      = Just <$> p <|> succeed Nothing
```

The order in which we denote the attributes determines the order in which the results are returned. Therefore, we can apply the *Img* constructor to form a value of the *XHTML* data type. The helper function *field* is used to parse a single attribute.

```
field       :: Parser p => String -> p a -> p a
field s p = token s  *> symbol '='  *> p
```

2.4.2 Haskell's record syntax

Haskell allows data types to contain labelled fields. If one wants to construct a value of that data type, one can make use of these names. The advantage is that the user does not need to remember the order in which the fields of the constructor have been defined. Furthermore, all fields are considered as optional. If a field is not explicitly set to a value, it is silently assumed to be $\bot$.

Whereas compilers support order-free syntax (the record fields in a program can be ordered arbitrarily), the *read* function expects the field in the same order as in the data type declaration. The resulting asymmetry is unfortunate. Using the *permuteSep* combinator, it is an easy task to write more flexible parsers for data types with labelled fields.

```
justOrNothing     :: Parser p => p a -> p (Maybe a)
justOrNothing p   = Just <$  token "Just" <*> p
                  <|> Nothing <$  token "Nothing"
img               :: ParserSplit p => p XHTML
img               = token "Img"  *> token "{"  *> fields <*  token "}"
  where fields    = permuteSep (symbol ',') $
    Img <$$> recordfield "src"       uri
        <|> recordfield "alt"        string
        <|> recordfield "longdesc"  (justOrNothing uri)
        <|> recordfield "height"    (justOrNothing int)
        <|> recordfield "width"     (justOrNothing int)
```

We use *recordfield* here to parse a single optional record field, returning $\bot$ if it is not present.

```
recordfield     :: Parser p => String -> p a -> p a
recordfield f p = field f p <|> succeed ⊥
```

2.5 Conclusion

We have shown how to extend a parser combinator library with the functionality to parse free-order constructs. It can be placed on top of any combinator library that implements the *Parser* interface. A user of the library can easily write parsers for free-order constructs and does not need to care about checking and reordering the parsed elements. Due to the use of existentially quantified types the implementation of reordering is type safe and hidden from the user.

The underlying parser combinators can be used to handle errors, such as missing or duplicate elements, since the extension inherits their error-reporting or error-repairing properties.

We have shown how our extension can be used to parse XML attributes and Haskell records. Other interesting examples mentioned by Cameron [9] include citation fields in BibTeX bibliographies and attribute specifiers in C declarations. Cameron's pseudo-code algorithm uses a similar strategy. It does not show, however, how to maintain type safety by undoing the change in semantics resulting from reordering, nor can it deal with the presence of separators between free-order constituents.

Chapter 3

Typing Dynamic Typing

This is a slightly edited version of a paper originally published as:
Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

Typing Dynamic Typing

Arthur I. Baars and S. Doaitse Swierstra
Institute of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands

Abstract: Even when programming in a statically typed language we every now and then encounter statically untypable values; such values result from interpreting values or from communicating with the outside world. To cope with this problem most languages include some form of *dynamic* types. It may be that the core language has been explicitly extended with such a type, or that one is allowed to live dangerously by using functions like *unsafeCoerce*. We show how, by a careful use of existentially and universally quantified types, one may achieve the same effect, without extending the language with new or unsafe features. The techniques explained are universally applicable, provided the core language is expressive enough; this is the case for the common implementations of Haskell. The techniques are used in the description of a type checking compiler that, starting from an expression term, constructs a typed function representing the semantics of that expression. In this function the overhead associated with the type checking is only once being paid for; in this sense we have thus achieved static type checking.

3.1 Introduction

For a statically typed programming language the typing rules are checked at compile-time. Type violations are reported before program execution, and efficient object code can be generated since no type consistency checks have to be performed at run-time. Even when using statically typed languages however, the need arises to deal with values with types that cannot be determined at compile-time. This situation occurs for example in a program that interprets a (meta)language term, resulting in an object language value of a type that depends on the specific term at hand. An example of such an interpretation function is *eval* which takes a string, parses it into an expression and returns the value of that expression. Other examples where type information is not available until run-time are distributed programs, that exchange data between different processes, and programs that store a value of an arbitrary type in, and retrieve it from stable storage.

There exist several proposals[1, 2, 40] for dealing with such dynamically typed values in a statically typed language. These are all based on a similar idea: *extend the language* with a universal type *dynamic*, and embed dynamic values in that type. This new type actually is a pair consisting of the value itself,

together with a representation of the type of that value. Before using a dynamic value its type component is inspected using a *typecase* or similar construct.

The proposals differ in their capability to deal with polymorphic dynamic values. In [1] dynamic polymorphism is forbidden, for [40] a restricted form of dynamic polymorphism is allowed and the system described in [2] allows polymorphic values with little restrictions, and as to be expected is also the most complex one. The latest release of Clean[51] implements support for dynamic typing[50] by providing a typecase construct and allowing polymorphic dynamic values. A drawback noted by Shields *et al.* [55] is that types live in two different worlds, with explicit conversions between these two worlds; he views dynamic typing as staged type inference. Some program expressions thus have their type inference deferred until enough information is available at run-time.

Despite ongoing research on dynamic typing, many statically typed languages – such as Haskell[46] – do not have built-in support for dynamic typing. A programmer thus must encode the dynamic type tagging and type checks explicitly. If the set of types is finite and known at compile-time, one may decide to embed values in a user-defined data type, and subsequently inspect the types of a values using case-analysis. With this approach, however, programs are difficult to maintain and become verbose and messy because type-checking code is intertwined with normal code.

To alleviate this problem the distributions of both Hugs and GHC provide a dynamic typing library. This library provides a universal representation *Dynamic* for dynamically typed values. Furthermore it provides a function *toDyn* that injects a value of arbitrary type into a dynamically typed value and a function *fromDyn* that converts a dynamic value into a concrete (monomorphic) type. Although this library cannot deal with polymorphic dynamic values, it has proven to be quite useful in practice. It provides a cheap way to program with dynamically typed values in Haskell. The implementation of this library is fairly simple and is based on the function *unsafeCoerce* which has type $(\forall ab.a \to b)$. As the name suggests this function is not type-safe, a direct consequence of the fact that safe functions of type $(\forall ab.a \to b)$ cannot exist at all.

In this paper we develop a statically typable library that provides a form of dynamic typing, that is as powerful as Haskell's Dynamic library, but does not make use of unsafe functions or compiler extensions. The attractive aspect of this approach is that we dot not extend the language itself, nor its implementation. Thus the technique can be universally applied, provided the type system of the host language is powerful enough to type our library, as is the case for Haskell with its common extensions for existentially and universally quantified types and constructors.

This paper is organized as follows: Section 3.2 introduces the concepts of dynamic typing. In Section 3.3 a data type is developed that serves as a witness of a proof that two types are equal. Section 3.4 defines a set of proof combinators to easily construct inhabitants of this data type. A key ingredient for dynamic typing are run-time type tags, and thus in Section 3.5 a class of type representations is introduced and a typical instance of this class is given. In Section 3.6 an interpreter for expression terms is described as an application of

the techniques developed in this paper. Section 3.7 concludes.

3.2 Dynamic Typing

Similar to Haskell's dynamic library and other approaches to dynamic typing we take the data type *Dynamic* to be a pair of a value together with the representation of its type. Suppose we have a type *typeRep* that is used to represent types; then the question arises how we can convince the type checker of our core language (in our case Haskell), that a value of type *typeRep* corresponds to a specific Haskell type; i.e. how can we label such a type representation with the type it represents. The answer lies in using a type constructor instead of a simple type, i.e. by passing the label type as an argument to the representation of the type.

A dynamic value can be viewed as a box containing a value of some type and the representation of that type. Since we do not yet want to fix the type we use for representing types, we postpone this decision by making this *typeRep* a parameter of the data type *Dynamic*. The actual type of the value injected in a *Dynamic* is hidden by existentially quantifying[1] over its type. Note that ::: is an infix constructor function with two arguments, namely an a and a *typeRep* a.

$$\textbf{data}\ \mathit{Dynamic}\ \mathit{typeRep} = \exists a\ .\ a ::: \mathit{typeRep}\ a$$

Using the ::: constructor a value can be packed as a dynamic, hiding the actual type of the value.

$$\begin{aligned} &\mathit{toDynamic} && :: a \to \mathit{tpr}\ a \to \mathit{Dynamic}\ \mathit{tpr} \\ &\mathit{toDynamic}\ a\ \mathit{tpr\_a} && = a ::: \mathit{tpr\_a} \end{aligned}$$

A value with an existential type can be unpacked using a case-construct.

$$\begin{aligned} &\textbf{case}\ \mathit{dynval}\ \textbf{of} \\ &\quad a ::: \mathit{tpr\_a} \to \ldots \end{aligned}$$

A skolem constant is introduced to serve as a placeholder for the type of a and the label of the type of the type representation *tpr*. Unfortunately, the value a cannot simply be returned, because then the skolem constant representing the type of a would escape, i.e appear in a type outside the scope of the case-expression.

We thus need a partial function *fromDyn* that converts a dynamic value into a value with a concrete type:

$$\mathit{fromDyn} :: \mathit{typeRep}\ a \to \mathit{Dynamic}\ \mathit{typeRep} \to \mathit{Maybe}\ a$$

[1] The implementors of the Haskell extensions have chosen to denote this existential quantifier with a **forall** keyword. In this presentation we have taken the liberty to use a somewhat more intuitive notation using the ∃ symbol

The function *fromDyn* takes two arguments, the representation of the expected type and a dynamic value. Dynamic type checking now boils down to comparing the type representation of the expected type, say *t1* of type *tp a* and the representation of the dynamic value's type, say *t2* of type *tp b*, and to make sure that the following equivalence holds: $(t1 \sim t2) \leftrightarrow (a \equiv b)$, i.e. structural equivalence of two type representations implies type equivalence of their labels and vice versa. Since the right to left direction is not used we will focus on the left to right direction of this implication, i.e. on $(t1 \sim t2) \rightarrow (a \equiv b)$. The function *fromDyn* must compare the representations of the expected type and the dynamic's type tag. If they are equal then the value stored in the dynamic is returned as a value of the expected type. Unfortunately simply returning the value stored in the dynamic will not work: the type of this value is unknown since it is still shielded by the existential quantification. The question thus arises how to convince Haskell's type-checker of the fact that a successful comparison implies that the hidden type equals the expected type, and hence is not unknown.

3.3 Type Equality

In this section we develop a data type that serves as evidence for the type-checker that two types are equal.

In a first approach we consider two types equivalent if the first can be converted into the second and vice versa, as expressed in by the type *Equal*:

$$\textbf{type}\ \mathit{Equal}\ a\ b = (a \rightarrow b, b \rightarrow a)$$

This encoding of type equivalence is also used by Yang[68] and Weirich[67]. Interestingly, this type is the encoding, according the Curry-Howard isomorphism of the following definition of equivalence on propositions. Note that function arrows map to implications and pairs to conjunctions.

$$a \equiv b = a \rightarrow b \wedge b \rightarrow a$$

The Curry-Howard isomorphism concerns the correspondence between logical formulas and types, in which a logical proof corresponds to a computational term. The existence of a (non-diverging) term of a certain type implies the existence of a proof of the corresponding logical formula, and vice versa. Hence the existence of an embedding-projection pair of type *Equal a b* can be considered a proof that *a* and *b* are equivalent.

Unfortunately the above solution fails to enforce that the existence of a value of type *Equal a b* implies that *a* and *b* are truly equal, as can be seen in the following example:

$$\begin{aligned}&\mathit{eqIntBool} :: \mathit{Equal\ Int\ Bool}\\ &\mathit{eqIntBool} = (\mathit{even}, \lambda x \rightarrow \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ 0)\end{aligned}$$

The problem is that arbitrary functions can be chosen as conversion function. So we want to restrict the type *Equal* in such a way that it is ensured that it can

only contain conversion functions that leave the dynamic value unchanged. This can be achieved by "shielding" the types a and b using a universally[2] quantified type constructor f:

$$\textbf{data}\ \mathit{Equal}\ a\ b = \mathit{Equal}\ (\forall f\ .\ f\ a \to f\ b)$$

This encoding of type equality, in a more implicit way, also serves as a basis for the implementation of a type-safe cast in Weirich[67].

Interestingly, the data type *Equal* is actually the encoding of Leibnitz' law which states that if a and b are identical then they must have identical properties. Leibnitz' original definition reads as follows

$$a \equiv b = \forall f\ .\ f\ a \Leftrightarrow f\ b$$

and can be proven to be equivalent to:

$$a \equiv b = \forall f\ .\ f\ a \to f\ b$$

The *Equal* data type encodes true type equality, since the identity function is the only non-diverging conversion function that can be used as argument of the *Equal* constructor. As the conversion function has to work for any f, it cannot make assumptions about the structure of f, making it impossible to construct a value of type $f\ a$ or to access values of type a that may be stored inside a value of type $f\ a$. Hence it is impossible for a conversion function to alter the value it takes as argument. Not taking into account the failing functions $\bot$ and $\lambda x \to \bot$, the identity function is the only function that can be used to construct a value of type *Equal*. The existence of a value of type *Equal a b* now implies that $a \equiv b$, since the conversion function, that converts an a into a b, must be the identity function.

In the definition of *Equal* the kinds of the type variables a and b can not be determined. Instead of defaulting these kinds to $*$, as is done in Haskell, we assume our language supports polymorphic kinds and assign the following kind to the type constructor *Equal*:

$$\mathit{Equal} : \forall k\ .\ k \to k \to *$$

Kind polymorphism would make a language such as Haskell more powerful and flexible. Consider, for example, the following type constructors *Int*, $[\,]$ and $\to$ which have kinds $*$, $* \to *$, and $* \to * \to *$ respectively. Because of the polymorphic kind the types *Equal Int Int*, *Equal* $[\,]$ $[\,]$ and *Equal* $(\to)$ $(\to)$ are all valid, whereas Haskell with its defaulting rule only accepts the first.

3.4 Building Equalities

Values of type *Equal a b* can be viewed as a proof that the types a and b are equal. This section introduces a set of proof combinators that can be used to

[2]Note that we have chosen to write the Haskell keyword **forall** as a $\forall$

easily construct equality proofs, and thus to create inhabitants of the *Equal* type.

We continue by showing that the type *Equal* implements an equivalence relation on types. The properties of an equivalence relation, namely reflexivity, transitivity and symmetry, can be encoded as:

Reflexivity $(a \equiv a)$:

$$\begin{aligned} &\mathit{reflex} :: \mathit{Equal}\ a\ a \\ &\mathit{reflex} = \mathit{Equal}\ \mathit{id} \end{aligned}$$

Transitivity $(a \equiv b \wedge b \equiv c \Rightarrow a \equiv c)$:

$$\begin{aligned} &\mathit{trans} :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ c \to \mathit{Equal}\ a\ c \\ &\mathit{trans}\ \mathit{ab}\ \mathit{bc} = \mathbf{case}\ (\mathit{ab}, \mathit{bc})\ \mathbf{of} \\ &\qquad\qquad (\mathit{Equal}\ f, \mathit{Equal}\ g) \to \mathit{Equal}\ (g\ .\ f) \end{aligned}$$

The implementation of *symmetry* $(a \equiv b \Rightarrow b \equiv a)$ is based on the following idea. If a and b are equal, we can freely substitute a's in a term by b's. Assuming $a \equiv b$ we can derive that $b \equiv a$ by starting with $a \equiv a$ (reflexivity) and applying the substitution $a \mapsto b$ on the first a.

The question that arises is how to implement such a substitution. Recall that $a \equiv b$ is encoded by a transform function of type $\forall f\ .\ f\ a \to f\ b$. It substitutes the argument a of a type constructor by b. In order to do a substitution on the a's in a type t, we first introduce a type constructor c, that is the abstraction of t over the a's we wish to substitute. The type t is equivalent to $c\ a$, applying the transformation function yields a result of type $c\ b$, which is the type t with the a's substituted by b's. For example we wish to apply the substitution $a \mapsto b$ on the type (a, a). This type is equivalent to the application $(\Lambda x.(x, x))a$. Applying the transform function yields a result of type $(\Lambda x.(x, x))b$, which can than be reduced to (b, b)

In Haskell lambda abstraction on type level can be mimicked using data types. The data type corresponding to the tuple example above is:

$$\mathbf{data}\ \mathit{Pair}\ x = \mathit{Pair}\ \{\mathit{unPair} :: (x, x)\}$$

A value of type (a, a) can be tagged with the *Pair* constructor to view its type as the application *Pair a*. After applying the transformation function, the reduction of the type *Pair b* to (b, b) is done by untagging the value. The function *substPair* is the implementation of the substitution $a \mapsto b$ on the type (a, a). It uses the function *subst*, that takes care of the tagging and untagging and performs the actual substitution by applying the transformation function *ab*.

$$\begin{aligned} &\mathit{subst} :: (t_a \to c\ a) \to (c\ b \to t_b) \to \mathit{Equal}\ a\ b \to t_a \to t_b \\ &\mathit{subst}\ \mathit{from}\ \mathit{to}\ (\mathit{Equal}\ \mathit{ab}) = \mathit{to}\ .\ \mathit{ab}\ .\ \mathit{from} \end{aligned}$$

$$\begin{aligned} &\mathit{substPair} :: \mathit{Equal}\ a\ b \to (a, a) \to (b, b) \\ &\mathit{substPair} = \mathit{subst}\ \mathit{Pair}\ \mathit{unPair} \end{aligned}$$

We will use the term type combinator for type constructors such as *Pair*. A type combinator constructs a type (in this case (a, a)) from its components(in this case a). The last argument of a type combinator is the target of the substitution. The others (if any) are unaffected by the substitution. Consider for example the type combinator *Middle*, which combines three component types into a triple:

$$\begin{aligned}&\textbf{data}\ \mathit{Middle}\ x\ y\ z = \mathit{Middle}\ \{\mathit{unMiddle} :: (x, z, y)\}\\&\mathit{substMiddle} :: \mathit{Equal}\ a\ b \to (x, a, y) \to (x, b, y)\\&\mathit{substMiddle} = \mathit{subst}\ \mathit{Middle}\ \mathit{unMiddle}\end{aligned}$$

As the middle component of the triple(z) is the last argument of the type combinator it is the target of the substitution, whereas the first and the third component are unaffected. Any substitution can be expressed using the function *subst* and the appropriate type combinator.

Finally we have all the ingredients to implement the symmetry property for the *Equal* type. The idea is that *Equal b a* can be derived from *Equal a b* and reflexivity (*Equal a a*) by applying the substitution $a \mapsto b$ to the first a in *Equal a a*. Firstly we define a type combinator *FlipEqual* to allow a substitution on the first argument of the *Equal* type constructor:

$$\textbf{data}\ \mathit{FlipEqual}\ y\ x = \mathit{Flip}\ \{\mathit{unFlip} :: \mathit{Equal}\ x\ y\}$$

The type variable x represents the target of the substitution since it is the last argument of *FlipEqual*. The symmetry property can now straightforwardly expressed:

Symmetry $(a \equiv b \Rightarrow b \equiv a)$:

$$\begin{aligned}&\mathit{symm} \quad :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ a\\&\mathit{symm}\ ab = (\mathit{subst}\ \mathit{Flip}\ \mathit{unFlip}\ ab)\ \mathit{reflex}\end{aligned}$$

The functions *reflex*, *trans*, and *symm* are proof combinators, they implement proof rules, and construct proofs out of other proofs. In the remainder of this section we extend our library of proof combinators with other rules. Substitutions play a key role in the implementation of these rules.

We proceed by introducing the following rule:

$$\textsc{Arg} : \frac{a \equiv b}{f\ a \equiv f\ b}$$

This rule is implemented by the function *arg*, which has the following type:

$$\mathit{arg} :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ (f\ a)\ (f\ b)$$

Notice that *Equal* $(f\ a)\ (f\ b)$ is encoded as a function with the following type $(\forall g\ .\ g\ (f\ a) \to g\ (f\ b))$. Thus the function *arg* can be implemented as a substitution on the a in the type $g\ (f\ a)$. The type combinator, that is required for this substitution, is the following:

$$\textbf{data}\ \mathit{Comp}\ g\ f\ x = \mathit{Comp}\ \{\mathit{unComp} :: g\ (f\ x)\}$$

Using this type combinator, the function *arg* is now defined as:

$$arg\ ab = Equal\ (subst\ Comp\ unComp\ ab)$$

A very useful proof combinator derived from *arg* is *rewrite*. It can be used to apply a substitution to the argument of a type constructor. For example if we have two proofs $x :: Equal\ a\ Int$ and $list :: Equal\ b\ [a]$ then we can construct a proof of type $Equal\ b\ [Int]$ as follows: $rewrite\ x\ list$.

$$\begin{aligned} &rewrite :: Equal\ a\ b \to Equal\ c\ (f\ a) \to Equal\ c\ (f\ b) \\ &rewrite\ a\_b\ c\_fa = \mathbf{let}\ fa\_fb = arg\ a\_b \\ &\qquad\qquad\qquad\qquad \mathbf{in}\ trans\ c\_fa\ fa\_fb \end{aligned}$$

The function *rewrite* can only apply a substitution to the last argument of a type constructor, such as the argument of the list type constructor or the result part of the function arrow. We now formulate the corresponding rule for the function part:

$$\textsc{Func} : \frac{f \equiv g}{f\ a \equiv g\ a}$$

The type of the function implementing this rule is:

$$func :: Equal\ f\ g \to Equal\ (f\ a)\ (g\ a)$$

Note that a value of type $Equal\ (f\ a)\ (g\ a)$ is represented as a function with the following type: $\forall h\ .\ h\ (f\ a) \to h\ (g\ a)$. Hence a substitution must be applied on the f in the type $h\ (f\ a)$. This substitution is again implemented using the function *subst* and an appropriate type combinator.

$$\mathbf{data}\ Haf\ h\ a\ f = Haf\ \{unHaf :: h\ (f\ a)\}$$

$$func\ x = Equal\ (subst\ Haf\ unHaf\ x)$$

Using *func*, it is easy to define a rewrite function that applies a substitution the one but last argument of a type constructor:

$$\begin{aligned} &rewrite' :: Equal\ a\ b \to Equal\ c\ (f\ a\ d) \to Equal\ c\ (f\ b\ d) \\ &rewrite'\ x\ y = trans\ y\ (func\ (arg\ x)) \end{aligned}$$

A rewrite function for the n but last argument of a type constructor can be obtained by applying *func* n times, as expressed in the following generic definition:

$$rewrite_n\ x\ y = trans\ y\ (fun\_n\ (arg\ x))$$

For example the rewrite function for the two but last argument is obtained by applying *func* twice:

$$rewrite_2 :: Equal\ a\ b \to Equal\ c\ (f\ a\ d\ e) \to Equal\ c\ (f\ b\ d\ e)$$
$$rewrite_2\ x\ y = trans\ y\ (func\ (func\ (arg\ x)))$$

Unfortunately, Haskell does not support polymorphic kinds, making the type of *func* invalid. Hence the generic solution for type constructors of different arities cannot be given, and instead, we must give specialized definitions for type constructors of different kinds. For example the function *rewrite′* for type constructors of kind $* \to * \to *$ can be defined as follows.

Firstly, we need a version of the type *Equal* specialized for type constructors of kind $* \to *$:

$$\begin{aligned}&\textbf{data}\ Equal'\ f\ g = Equal'\ (\forall h\ .\ h\ f \to h\ g)\\ &\qquad\qquad\quad \mid\ KindInfo\ (f\ ())\ (g\ ())\end{aligned}$$

In order to prevent Haskell's kind inferencer to default the kind of f and g to $*$, a dummy alternative[3] is provided, in order to force the inferencer to infer $* \to *$ instead.

In a similar way the composition type constructor is specialized, the dummy alternative enforces that g has kind $* \to * \to * \to *$:

$$\begin{aligned}&\textbf{data}\ Comp'\ f\ g\ x = Comp'\ \{unComp' :: f\ (g\ x)\}\\ &\qquad\qquad\quad \mid\ KindInfo2\ (g\ ()\ ())\end{aligned}$$

Using these specialized data types , specialized versions of the functions *arg* and *func* can be defined in a similar way as before:

$$\begin{aligned}&arg' \quad :: Equal\ a\ b \to Equal'\ (f\ a)\ (f\ b)\\ &arg'\ ab = Equal'\ (subst\ Comp'\ unComp'\ ab)\end{aligned}$$

$$\begin{aligned}&subst' :: (t_a \to c\ a) \to (c\ b \to t_b) \to Equal'\ a\ b \to t_a \to t_b\\ &subst'\ from\ to\ (Equal'\ ab) = to\ .\ ab\ .\ from\end{aligned}$$

$$\begin{aligned}&func' \quad :: Equal'\ f\ g \to Equal\ (f\ a)\ (g\ a)\\ &func'\ fg = Equal\ (subst'\ Haf\ unHaf\ fg)\end{aligned}$$

Finally, a valid definition for *rewrite′* can be given:

$$\begin{aligned}&rewrite' :: Equal\ b\ d \to Equal\ a\ (f\ b\ c) \to Equal\ a\ (f\ d\ c)\\ &rewrite'\ a\ b = trans\ b\ (func'\ (arg'\ a))\end{aligned}$$

Fortunately, specializations for more complex kinds are not required, as they are not used in the remainder of this paper.

Using the combinators defined above, the law of congruence for a binary functor:

$$\text{CONGRUENCE}: \frac{a \equiv b, c \equiv d}{f\ a\ c \equiv f\ b\ d}$$

[3]The latest release of GHC supports kind annotations. These provide a more elegant solution for assigning an explicit kind to a type variable, than the use of dummy constructors.

can be defined as follows:

```
congruence :: Equal a b → Equal c d → Equal (f a c) (f b d)
congruence ab cd = rewrite cd (rewrite' ab reflex)
```

Equivalence proofs can be constructed easily using the proof combinators defined thus far. Consider for example the following proposition and its proof:

$x \equiv a \to b, y \equiv c \to d, a \equiv c, b \equiv d \vdash x \equiv y$

1	$x \equiv a \to b$	(assumption)
2	$y \equiv c \to d$	(assumption)
3	$a \equiv c$	(assumption)
4	$b \equiv d$	(assumption)
5	$a \to b \equiv c \to d$	(congruence: 3, 4)
6	$c \to d \equiv y$	(symmetry: 2)
7	$x \equiv c \to d$	(transitivity: 1,5)
8	$x \equiv y$	(transitivity: 7,6)

This proof is easily encoded using proof combinators as follows:

```
deduce                  :: Equal x (a → b)
                        → Equal y (c → d)
                        → Equal a c
                        → Equal b d
                        → Equal x y
deduce x1 x2 x3 x4 = let x5 = congruence x3 x4
                         x6 = symm x2
                         x7 = trans x1 x5
                         x8 = trans x7 x6
                     in x8
```

3.5 Type Representations

Thus far, we have defined an encoding of type equivalence proofs and a set of combinators to construct such proofs. Since we do not want to fix the type used for type representations, we introduce the class *TypeDescr*, that only contains the function $(\sim)$ that is used for checking whether a type representation *tpr a* represents the same type as a type representation *tpr b*. The *Maybe* data type is used to distinguish between a successful comparison, in which case a witness of type *Equal a b* is returned, and an unsuccessful one, in which case *Nothing* is returned.

```
data Maybe a = Just a
             | Nothing

class TypeRep tpr where
  (∼) :: tpr a → tpr b → Maybe (Equal a b)
```

We continue by defining the function *coerce* for the *Equal* type, using the function *subst* and the identity type constructor:

```
data Id x = Id {unId :: x}

coerce     :: Equal a b → (a → b)
coerce ab = subst Id unId ab
```

Using the operator (∼) the function *fromDyn*, that converts a dynamic into a normal value, can now easily be expressed:

```
fromDyn :: TypeRep tpr ⇒ tpr a → Dynamic tpr → Maybe a
fromDyn et (x ::: t) = case t ∼ et of
                         Just eq → Just (coerce eq x)
                         Nothing → Nothing
```

The function *fromDyn* checks whether the expected type *et* matches the type of the value stored in its second argument. If this is the case the stored value is "converted", using the constructed evidence *eq*, into the expected type and returned. Note that the only way in which this conversion can be done, is by applying the conversion function returned by the call of (∼) in case the types match.

We continue by defining a data type *TpCon* that represents type constants, such as *Int* and *Bool*. The constructors *Int* and *Bool*, take a proof that the type-label *a* equals the type *Int* respectively *Bool*:

```
data TpCon a = Int (Equal a Int)
             | Bool (Equal a Bool)
```

Since we want to be able to compare such types we make it an instance of *TypeRep*. The function (∼) checks whether both its arguments are structurally equivalent and if this is the case constructs a proof that both arguments are of the same type using the transitivity and symmetry combinators.

```
instance TypeRep TpCon where
  (∼) (Int x) (Int y)   = Just (trans x (symm y))
  (∼) (Bool x) (Bool y) = Just (trans x (symm y))
  (∼) _       _         = Nothing
```

Smart constructors for the type representations of the types *Int* and *Bool* are defined as *inttp* and *booltp*:

```
inttp  :: TpCon Int
inttp  = Int reflex
booltp :: TpCon Bool
booltp = Bool reflex
```

Using these we can easily inject values of type *Int* and *Bool* together with their type representations into a *Dynamic*.

```
true       = True ::: booltp
ninetythree = 93 ::: inttp
```

Now the operation *fromDyn* can be used to try to convert dynamic values into a concrete type. For example: *fromDyn booltp true* results in *Just True* and *fromDyn booltp ninetythree* will fail.

Since we now not only want to deal with constant types we introduce a new type *TpRep* that extends the type representation with alternatives for list and function types, containing the representations of the constituting types:

```
data TpRep tpr a = TpCon (tpr a)
                 | ∃x . List     (Equal a [x])
                                 (TpRep tpr x)
                 | ∃x y . Func (Equal a (x → y))
                                 (TpRep tpr x)
                                 (TpRep tpr y)
```

For the new type descriptor *TpRep* the smart constructors of the representations of *Int* and *Bool* need to be redefined:

```
type Type = TpRep TpCon


inttp  :: Type Int
inttp  = TpCon (Int reflex)
booltp :: Type Bool
booltp = TpCon (Bool reflex)
```

The smart constructor *list* for list types takes as an argument the representation of the component type:

```
list         :: TpRep tpr a → TpRep tpr [a]
list tpr_a = List reflex tpr_a
```

For function types the right-associative operator .→. is defined, that constructs a function type representation out of its argument and result type representation.

```
(.→.) :: TpRep tpr a → TpRep tpr b → TpRep tpr (a → b)
a .→. r = Func reflex a r
```

The smart constructors provide a convenient notation for constructing type representations; for example the representation for $[Int] \to Bool$ can be expressed as follows: *list inttp .→. booltp*.

By now making *TpRep* an instance of the class *TypeDescr* it becomes possible to cast function types and list types.

In order to match two type constants the function (∼) on the underlying representation of type constants is called.

```
instance TypeRep tpr ⇒ TypeRep (TpRep tpr) where
  (∼) (TpCon x) (TpCon y) = x ∼ y
```

For list types the component types are compared, resulting in an equivalence proof of both component types if the comparison succeeds. This proof is subsequently used to construct the proof that both list types match.

```
(∼) (List x t1) (List y t2) = case t1 ∼ t2 of
  Just eq  → Just (trans (rewrite eq x) (symm y))
  Nothing → Nothing
```

The definition of (∼) on two function types is quite straight-forward: if we can prove that their argument types are equal and we can prove that their result types are equal then we may deduce that both function types are equal.

```
(∼) (Func x a1 r1) (Func y a2 r2) =
  case (a1 ∼ a2, r1 ∼ r2) of
    (Just arg, Just res) → Just (deduce x y arg res)
    _                    → Nothing
```

Finally, if the type representations do not match then no equivalence proof can be constructed, hence *Nothing* is returned.

```
(∼) _ _ = Nothing
```

A function value can be cast into a dynamic value, and later on be down-cast to a value of its original type. During a down-cast a type-check is performed once, and afterwards we can use the function many times without again paying for type-checking. Consider for example the function *plus* and the value *one*, resulting from an up-cast of the operator (+) and the integer 1, respectively:

```
plus :: Dynamic Type
plus = (+) ::: inttp .→. inttp .→. inttp


one :: Dynamic Type
one = 1 ::: inttp
```

Two values can be applied, if the first one has a function type and the type of other matches the argument type of the function:

```
dynApply :: TypeRep tp ⇒ Dynamic (TpRep tp)
                       → Dynamic (TpRep tp)
                       → Maybe (Dynamic (TpRep tp))
dynApply (f ::: ft) = case ft of
  Func eqf arg res
    → let f' = coerce eqf f
      in  λ(x ::: xt) →   case xt ∼ arg of
                            Just eqa → let x' = coerce eqa x
                                       in  Just (f' x' ::: res)
                            Nothing → Nothing
  _  → const Nothing   -- not a function type
```

The value *inc* is the result of the application of *plus* to *one*:

```
inc :: Dynamic Type
inc = case dynApply plus one of
        Just v → v
```

The value *inc* contains the function ((+) 1) together with a description of its type, which is (*Int* → *Int*). The function *increment* is the result of a down-cast of *inc*. After the down-cast we end up with a function that adds 1 to a number without performing the type-check again.

```
increment :: Int → Int
increment = case fromDyn (inttp .→. inttp) inc of
              Just f → f
```

We want to stress once more that inside the expressions that are constructed there initially live many applications of identity functions to values. When our function is evaluated for the first time, all such applications of identities will be removed, and the overhead resulting from type checking the expression when building it is gone.

3.6 Interpreting Expressions

In this section we show how to use the machinery from the previous section in constructing efficient typed evaluators. We do so in the context of a syntax macro mechanism, in which an already existing parser for some language is dynamically extended to recognize and process a larger language. A lot of research has been done on syntax macros and related topics[37, 49, 10, 56]. Our notation for syntax macros is borrowed from Cardelli *et al.*[10].

A syntax macro enabled compiler first reads a set of macro definitions – and after having processed them successfully– reads the actual program in the extended language. The semantics associated with the new language constructs is expressed in terms of expressions of a base language, and are in our case built from constructors describing the abstract syntax of the base language.

Since these definitions are parsed, processed and evaluated by the existing compiler we want to make sure that after successfully processing the macro definitions, we do not pay (much) more than when we had added the processing of the new constructs directly to the existing compiler. Furthermore we want to make sure that the compiler, after having successfully processed the macro definitions, does not break down due to a typing error when processing the actual program.

As an example we take a very simple language with the following abstract syntax:

```
data Exp = Add Exp Exp
         | Sub Exp Exp
         | IntLit Int
```

A set of syntax macros that defines a concrete syntax for this language typically looks as follows:

```
Expr, Factor :: Exp'
FactIter     :: Exp -> Exp'
Oper         :: Exp -> Exp -> Exp'
'
Expr     ::= x=Factor'
             fs=FactIter => fs x'
           | x=Factor    => x'
'
FactIter ::= op=Oper y=Factor'
             fs=FactIter => \x:Exp . fs (op x y)'
           |             => \x:Exp . x'
'
Oper     ::= "+"         => Add'
           | "-"         => Sub'
'
Factor     ::= x=IntLiteral  => IntLit x'
```

The macros start with the declaration of the types of new concrete nonterminals that are introduced by the macros. The constants in this type language are the nonterminals from the abstract syntax.

The declarations are followed by a list of production rules. The part of a production rule left of the arrow resembles a BNF production and defines the concrete syntax of the language. The right-hand side of a production rule defines a mapping to the abstract syntax of the core language. The parser constructed for the above macros recognizes expressions containing left associative $+$, $-$ operators and integer literals, and constructs a parse tree of type *Exp*.

Our syntax macro system constructs parsers at run-time using self-optimizing parser combinators[59] and calls this parser directly on a source file. It is not necessary to generate parse-tables off-line to improve efficiency as is common in most other implementations.

The idea behind combinator parsers is that a parser is a function that consumes a list of symbols and produces an abstract syntax tree of some type. The problem that we have to deal with now is that parsers must be well-typed just as any other function. Thus the macro system interprets a macro and has to construct a well-typed parser. In order to achieve this, run-time type-checking is required. Using the type-checking approach introduced before, well-typed parsers can be constructed in such a way that the cost of type-checking is paid for only once.

To illustrate the use of run-time type-checking during interpretation of the macros we show how the interpretation of the right-hand sides of the macro is done.

The abstract syntax of the expressions at the right-hand side of a production rule is defined by the following data type:

$$
\begin{aligned}
&\textbf{type } \mathit{Ident} &&= \mathit{String}\\
&\textbf{data } \mathit{Expression} &&= \mathit{Var}\quad \mathit{Ident}\\
&&&\mid \mathit{Apply}\ \mathit{Expression}\ \mathit{Expression}\\
&&&\mid \mathit{Const}\ (\mathit{Dynamic}\ \mathit{Type})\\
&&&\mid \exists x\ .\ \mathit{Lambda}\ \mathit{Ident}\ (\mathit{Type}\ x)\ \mathit{Expression}\\
&&&\mid \exists x\ .\ \mathit{Let}\quad \mathit{Ident}\ (\mathit{Type}\ x)\ \mathit{Expression}\ \mathit{Expression}
\end{aligned}
$$

Underlying the type checking is the concept of type judgments:

$$\Gamma \vdash \mathsf{expr} : t$$

We read this as follows: under the assumptions about the types of the variables in Γ, the expression *expr* is well typed and has type t.

The typing rule for constant values, like 3 and *True*, is trivial. It states that a constant of type *tp* has type *tp*:

$$\textsc{Constant} : \Gamma \vdash \mathsf{c_{tp}} : tp$$

Typing an expression consisting of a single identifier is described by:

$$\textsc{Ident} : \frac{(\mathsf{id} : t) \in \Gamma}{\Gamma \vdash \mathsf{id} : t}$$

The rule for function application is:

$$\textsc{Apply} : \frac{\Gamma \vdash \mathsf{expr_1} : a \to r, \Gamma \vdash \mathsf{expr_2} : a}{\Gamma \vdash \mathsf{expr_1\ expr_2} : r}$$

The rule for lambda abstraction is:

$$\textsc{Lambda} : \frac{(\Gamma - \mathsf{id}) \cup \{\mathsf{id} : a\} \vdash \mathsf{expr} : b}{\Gamma \vdash (\lambda \mathsf{id}{:}a\ .\ \mathsf{expr}) : a \to b}$$

Finally, the rule for let expressions is:

$$\textsc{Let} : \frac{(\Gamma - \mathsf{id}) \cup \{\mathsf{id} : a\} \vdash \mathsf{expr_1} : a, (\Gamma - \mathsf{id}) \cup \{\mathsf{id} : a\} \vdash \mathsf{expr_2} : b}{\Gamma \vdash (\textbf{let}\ \mathsf{id}{:}a = \mathsf{expr_1}\ \textbf{in}\ \mathsf{expr_2}) : b}$$

In our compiler we distinguish a compile-time and a run-time environment. The compile-time environment is used during type-checking and compilation, it is a symbol table containing the type and the location in the run-time environment for each variable. The run-time environment is used during evaluation of an expression and contains the actual values for the variables in the expression.

Nested pairs are a very suitable data structure for the run-time environment. They can store heterogenous data, which is important because the variables in an expression may have different types. Further more selector functions to access each value can easily be constructed using the function *fst* and *snd*. Consider for example an environment containing the values *True*, 3, and `'a'`:

$$(\mathit{True}, (3, \texttt{'a'}))$$

The selector functions for the first, second and third value are respectively *fst*, *fst* . *snd*, and *snd* . *snd*.

Compiled expressions need a run-time environment to provide values for their variables. Hence a compiled expression is represented as a function of type $(env \rightarrow a)$, that computes the value of the expression of type a, given an environment of type *env*, containing a valuation for its variables. As the result type of a compiled expression is dynamic, the function representing the expression is wrapped as a dynamic value.

The data type *DynamicF* is a generalization of *Dynamic*.

$$\textbf{data}\ DynamicF\ tpr\ f = \exists x\ .\ f\ x :::: tpr\ x$$

The functions for converting dynamic values into normal values can be defined for the type *DynamicF* in a similar way as done for the type *Dynamic*:

$$\begin{aligned}
&coerceF :: Equal\ a\ b \rightarrow f\ a \rightarrow f\ b\\
&coerceF\ (Equal\ eq) = eq
\end{aligned}$$

$$\begin{aligned}
&fromDynF :: TypeRep\ tpr \Rightarrow tpr\ a \rightarrow DynamicF\ tpr\ f \rightarrow Maybe\ (f\ a)\\
&fromDynF\ e\ (x :::: t) = \textbf{case}\ t \sim e\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Just\ eq\ \rightarrow Just\ (coerceF\ eq\ x)\\
&\qquad\qquad\qquad\qquad Nothing \rightarrow Nothing
\end{aligned}$$

The type of compiled expressions is defined by parametrizing *DynamicF* with the type constructor $((\rightarrow)\ env)$:

$$\textbf{type}\ CompExp\ env = DynamicF\ Type\ ((\rightarrow)\ env)$$

The compile-time environment is a symbol table storing for each variable its type and a the location of the variable at run-time. The type of an identifier is encoded as a type representation of type *Type* t, and the location as a selector function of type $(env \rightarrow t)$. The identifier information consisting of a type representation and a selector function are packed as a dynamic value:

$$\textbf{type}\ IdentInfo\ env = DynamicF\ Type\ ((\rightarrow)\ env)$$

The type of the symbol table is defined as:

$$\textbf{type}\ SymTable\ env = [(Ident, IdentInfo\ env)]$$

Each time a declaration of variable is introduced, a new entry with the type and selector function of the variable must be added to the symbol table. Furthermore, the type of the run-time environment has to change, since it has to store a value for the new identifier. The function *add* takes a pair of an identifier and the representation of its type and extends the environment with this new identifier.

$$
\begin{aligned}
&add :: (Ident, Type\ a) \to SymTable\ env \to SymTable\ (a, env)\\
&add\ (x, tp)\ gamma \qquad\qquad = (x, fst :::: tp) : map\ f\ gamma\\
&\quad \textbf{where}\ f\ (ident, get :::: t) = (ident, (get\ .\ snd) :::: t)
\end{aligned}
$$

In order to hold a value of type a for the new identifier, the type of the run-time environment is changed to a pair of a value of type a and the old environment. The selector function for the new identifier is obviously the function *fst*. As the type of the run-time environment changed from *env* to (a, env), the selector functions for the other identifiers must be composed with *snd*.

We proceed by developing a compiler for the simple expression language defined above. As the structure of the compiler function closely follows the typing rules, we choose it to resemble the type judgement operator:

$$(|-) :: SymTable\ env \to Expression \to Maybe\ (CompExp\ env)$$

This operator takes an environment and an expression as arguments; if all variables occurring in the expressions are introduced in the environment and there are no type errors then the compilation succeeds and returns the value of the expression as a compiled expression. The monadic **do** notation is used in the definition of $(|-)$ to deal with the *Maybe* type. It avoids cluttering the code with pattern matches on *Nothing* and *Just* and makes the resemblance of the implementation to the typing rules more clear.

The definition of the compile operator for constants is straightforward. No environment is needed to evaluate a constant, so the environment is discarded by the function *const* and the value of the constant is returned.

$$gamma\ |-\ Const\ (c ::: tp) = return\ (const\ c :::: tp)$$

The interpretation of function applications closely follows the corresponding typing rule. The first three lines of the **do** statement correspond to the three assumptions in the typing rule, whereas the last line builds the conclusion. The coerce functions *funEQ* and *argEQ* are applied in order to convince the typechecker that *e1* is a function and the type of *e2* matches the argument type of this function. In the resulting expression the environment *env* is passed to both expressions.

$$
\begin{aligned}
&gamma\ |-\ Apply\ expr1\ expr2\ =\\
&\quad \textbf{do}\ e1 :::: Func\ funEQ\ a\ r \ \leftarrow gamma\ |-\ expr1\\
&\qquad e2 :::: b \qquad\qquad\quad \leftarrow gamma\ |-\ expr2\\
&\qquad argEQ \qquad\qquad\qquad \leftarrow b \sim a\\
&\qquad \textbf{let}\ e1' \qquad\qquad\qquad = coerceF\ funEQ\ e1\\
&\qquad\quad e2' \qquad\qquad\qquad = coerceF\ argEQ\ e2\\
&\qquad return\ ((\lambda env \to e1'\ env\ (e2'\ env)) :::: r)
\end{aligned}
$$

For the interpretation of variables, the identifier is located in the environment *gamma*, yielding the representation of its type and a function that selects its value from the run-time environment:

```
gamma |- Var x = do var :::: tp <- lookup x gamma
                    return (var :::: tp)
```

The interpretation operation for lambda-abstractions adds the identifier with its type to the symbol table and uses the extended symbol table to compile the body of the lambda. At run-time the value for the identifier is placed into the run-time environment, which is subsequently used to compute the value of the body.

```
gamma |- Lambda x tp expr =
  do e :::: etp          <- add (x, tp) gamma |- expr
     let mkLam env = λx -> e (x, env)
     return (mkLam :::: tp .->. etp)
```

The compilation of a let expression proceeds in a similar way. During compilation the symbol table is extended with the identifier and type of the declaration. This symbol table is than used to compile both the declaration and the body. At run-time the value of the declaration is put into the environment, which is used to evaluate both the declaration and the body of the let expression.

```
gamma |- Let x tp expr1 expr2 =
  do let gamma'    = add (x, tp) gamma
     e1 :::: tp1   <- gamma' |- expr1
     e2 :::: tp2   <- gamma' |- expr2
     eq            <- tp1 ~ tp
     let mkLet env = let env' = (decl, env)
                         decl = coerceF eq e1 env'
                     in e2 env'
     return (mkLet :::: tp2)
```

Using the compiling operator we can compile expression terms into their values:

```
compile :: Expression -> Type a -> Maybe a
compile expr t = do res <- [] |- expr
                    val <- fromDynF t res
                    return (val ())
```

The function *compile* takes an expression term and starts with the empty environment ([]). If the resulting value matches the expected type an empty run-time () environment is supplied, and the value of the expression is returned.

As a final example now consider the expression corresponding to the lambda-term $\lambda x :: Int\ .\ \lambda y :: Int \rightarrow x + y$:

```
expr :: Expression
expr = (Lambda "x" inttp
          (Lambda "y" inttp
            (Apply
              (Apply
```

```
                (Const plus)
                (Var "x")
            )
            (Var "y")
        )
    )
)
```

This term is compiled into a function with type $Int \to Int \to Int$ as follows:

```
test :: Int → Int → Int
test = case compile expr (inttp .→. inttp .→. inttp) of
       Nothing → error "type error in expression"
       Just x  → x
```

The function *test* can now be applied to add two integers, without any further type checking being involved!

The expression compiler can also deal with recursive definitions as illustrated by the following example. Consider the following expression containing a recursive definition.

```
let ones = 1 : ones in ones
```

This expression is encoded as a term (*expr2*) of type *Expression*, using the helper functions *cons* and *int* to wrap the operator (:) and the integer value, respectively.

```
cons  :: Type t → Expression
cons t = Const ((:) ::: t .→. list t .→. list t)


int   :: Int → Expression
int x = Const (x ::: inttp)


expr2 :: Expression
expr2 = Let "ones" (list inttp)
                   (Apply (Apply (cons inttp) (int 1))
                          (Var "ones")
                   )
                   (Var "ones")
```

The expression *expr2* can be compiled to a Haskell value of type $[Int]$ as follows:

```
test2 :: [Int]
test2 = case compile expr2 (list inttp) of
          Nothing → error "type error in expression"
          Just x  → x
```

3.7 Conclusions

We have shown that extending a language with a separate dynamic typing mechanism is not needed, provided the core language is sufficiently rich. The techniques presented here can easily be incorporated within a small module. Our solution does not require large compiler modifications or ad hoc language extensions such as *unsafeCoerce*. The presented approach is type safe and as powerful as Haskell's dynamic typing library. In contrast to some of the other approaches to dynamic typing, our library does not support polymorphic dynamic values. Whether our approach can easily be extended with dynamic polymorphism is as yet unknown and a subject of further research.

In order to use the dynamic typing library an instance of the class of type representations must be provided. The code for these instances is completely generic and can be easily generated by a special tool; or by an extended version of the Generic Haskell compiler [16, 17]. Another solution is to let the compiler construct the type representations as an abstract data type. This opens the way for dynamic linking and reflection, e.g. by exporting from each module its environment in the form of a function of the type:

$$environment :: FiniteMap\ String\ Dynamic$$

In this way run-time access to the module's interface (`.hi`-file) is obtained. Occurrences of global variables in the expression terms that are compiled by the expression interpreter developed in the previous section could in that case be linked to the corresponding functions in the environment.

We have introduced a data type *Equal* to represent genuine type equality. It finds its theoretical basis in Leibnitz' definition of equality and the Curry-Howard isomorphism. We have developed a small library of proof combinators to constructs of the *Equal* type. Values of the type *Equal* are represented by identity functions. By combining these with function application or composition, the proof combinators construct new values of the *Equal* type. The proof combinators correspond to proof rules and make up a small proof system.

A dynamic value is cast to a concrete type by comparing the type of the dynamic value with the expected type, resulting in a proof that they are equal if they are. This proof is a composition of identity functions. The identity functions are applied to the dynamic value, coercing it to a value of the expected type. After the dynamic type checking overhead has been computed once, the resulting value is free of typing overhead and can subsequently be used just as a normal value.

Chapter 4

Type-safe Self-inspecting Code

This is a slightly edited version of a paper originally published as:
Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

Type-safe Self-inspecting Code

Arthur I. Baars and S. Doaitse Swierstra
Institute of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands

Abstract: We present techniques for representing typed abstract syntax trees in the presence of observable recursive structures. The need for this arose from the desire to cope with left-recursion in combinator based parsers. The techniques employed can be used in a much wider setting however, since it enables the inspection and transformation of any program structure, which contains internal references. The hard part of the work is to perform such analyses and transformations in a setting in which the Haskell type checker is still able to statically check the correctness of the program representations, and hence the type correctness of the transformed program.

4.1 Introduction

Embedding a domain specific language by means of a combinator library [60, 39] is common practice in Haskell. Examples of embedded domain specific languages include HaskellDB [39] for database programming, QuickCheck[14] for software testing, Wash/CGI[61] for server-side web applications, pretty printing[13, 29, 66, 48] and parsing combinator [20, 64, 33, 59, 38, 31] libraries.

An embedded domain specific language (EDSL) has many advantages over a normal domain specific language (DSL). First of all the design and implementation of an EDSL is easier. One does not have to implement an entirely new compiler, but can make use of the host language's compilers and tools. A EDSL inherits the abstraction mechanisms and typing system of the host language. Because EDSL programs are in fact programs in the host language we can easily combine different EDSL's in a single program.

A disadvantage of a EDSL is that it is constrained by the host language's syntax and type-system. Fortunately Haskell has a very powerful type system, and many notational features, such as monad-comprehensions (do-notation), operators and functions. However, when implementing a EDSL, one will very soon bump into the boundaries of Haskell's syntax or type system. Originally combinator based embedded languages directly expressed the denotational semantics of the embedded language. A logical next step along this line of development is to first build an intermediate structure, which can then be analyzed and transformed as in an ordinary compiler, thus getting the advantages from embedded compilers [60, 58]. For many analyses and transformations one really needs to get a hold the binding structure of the embedded program, especially when recursion is involved, since abstract interpretation has to deal with some form of

fixed-point computation in those cases.

In this paper we show how cycles in an object language term can be made observable, while maintaining a typed term representation. To achieve our goals we make use of the following Haskell extensions: existentially quantification, local universal quantification, and a single use of *unsafeCoerce* to implement an axiom for the equality data type. As a running example we take the parsing combinator library, that inspired this investigation. A problem with parsing combinators is that they cannot deal with left-recursive grammars. When a left-recursive grammar is transcribed straightforwardly into a combinator based parser, the resulting parser may not terminate. To be able to remove left-recursive cycles, while preserving type correctness, we employ the following individual techniques:

- Typed abstract syntax trees[45]
- Modelling of recursion using a custom fix-point combinator
- Left-recursion removal transformations

By making use of typed abstract syntax, not only the correctness of the object language (parsing combinators) is ensured by type correctness in the meta-language (Haskell), but, by staying in a typed world when performing transformations, we also provide a partial correctness proof of these transformations.

By using a custom fix-point combinator instead of meta-language recursion we make the cycles in the object language observable. This technique is similar to the translation described in the Haskell Report of **let**-expressions into a lambda abstraction and a call to a fix-point operator, or the de-sugaring of the **mdo**-construct[19] for recursive monads.

This paper is organized as follows: Section 4.2 describes the problem of left-recursive combinator parsers. Section 4.3 introduces the concept of meta-programming with typed abstract syntax. Furthermore it introduces the equality types, a key ingredient for typed abstract syntax. The equality type actually records the proofs that our transformations are type preserving. Section 4.4 shows the interface of the parsing combinators and a simple implementation. Section 4.5 describes the actual analysis of a grammar. This proceeds as follows, the grammar is analyzed for left-recursion; if it is left-recursive, it is transformed to remove the left-recursion. Finally, the transformed grammar is compiled into a real parser by mapping it onto normal parsing combinators. The programs manipulating the abstract syntax of the parsers are guaranteed to preserve type correctness, hence the title "Type-safe Self-Inspecting Code". Section 4.6 describes how to construct grammars objects using combinators that resemble parsing combinators. Unfortunately some of the notational elegance will be lost, mainly due to the use of a custom fix-point combinator instead of **let** or **mdo** constructs. Section 4.7 discusses some possibilities to solve this problem. Finally Section 4.8 concludes.

4.2 Problem

Attractive as combinator based parser construction may appear, there is one large problem looming over this approach. If a programmer translates a left-recursive grammar straightforwardly into a combinator based parser, the resulting parsers will not terminate, due to the top-down parsing strategy that is being employed.

In parser generators that generate a top-down based parser this problem can be alleviated by first analyzing the grammar, and transforming it in case the grammar is left-recursive. Unfortunately generating a parser offline, using a separate system, also brings its disadvantages, such as a usually not straightforward integration with semantic processing, the inability to extend a generated parser afterwards, and the need to implement a separate abstraction mechanism.

The problem however becomes more pressing if one wants to dynamically combine grammars that come from different places . In such cases it will not be attractive at all to perform explicit left-recursion removal or to rewrite the grammar, since this destroys the compositional nature of the code.

An example of this is in the implementation of some syntax macro mechanism ([37]). Here the programmer is allowed to extend the concrete syntax of his programming language, and it would be unreasonable to ask him to refrain from inadvertently constructing a left-recursive grammar; it is quite possible that he extends the grammar without even being aware of how large parts of the core grammar have been described or are being parsed. This problem is especially serious since there is no way to even warn him that the internally constructed grammar is left-recursive, and he will only discover the problem when the compiler reports a stack overflow, or a similar sign of internal unhealthiness shows up.

A similar situation arises in the use of the self analyzing parsers as introduced by Swierstra and Duponcheel ([58]), where a grammar is statically analyzed before a parser is constructed. A basic assumption for their approach is that the underlying grammar is *LL(1)*, but checking that this is indeed the case is not possible. For a full *LL(1)* check we need to compute the set of so-called *followers* for each non-terminal that may generate the empty string, but for this we need access to all applied positions of these nonterminals in the right hand sides of productions. Since we have only a very implicit representation of the grammar at hand we cannot get our hands on this information. So when using this library we may only discover that the underlying grammar is not *LL(1)* when we are running the parser on a concrete input, a quite unsatisfactory situation.

4.3 Meta-programming with typed abstract syntax

Meta-programming is about writing programs that manipulate other programs. In Pasalic' work [45] the meta-language Haskell is used to manipulate object-language terms, that are represented by Haskell data types. These data types are

$$\textsc{Int} : \vdash i : \mathbf{Int}$$

$$\textsc{Equals} : \frac{\vdash \mathsf{expr}_1 : \mathbf{Int}, \vdash \mathsf{expr}_2 : \mathbf{Int}}{\vdash \mathsf{expr}_1 \equiv \mathsf{expr}_2 : \mathbf{Bool}}$$

$$\textsc{Add} : \frac{\vdash \mathsf{expr}_1 : \mathbf{Int}, \vdash \mathsf{expr}_2 : \mathbf{Int}}{\vdash \mathsf{expr}_1 + \mathsf{expr}_2 : \mathbf{Int}}$$

$$\textsc{If} : \frac{\vdash \mathsf{expr}_1 : \mathbf{Bool}, \vdash \mathsf{expr}_2 : a, \vdash \mathsf{expr}_3 : a}{\vdash \mathbf{if}\ \mathsf{expr}_1\ \mathbf{then}\ \mathsf{expr}_2\ \mathbf{else}\ \mathsf{expr}_3 : a}$$

Figure 4.1: Typing judgments for *Expr* language

designed in such a way that context sensitive constraints of the object language are enforced by the Haskell type system that works for the term representation. This means that binding and typing errors in algorithms that manipulate terms of the object language are captured by the Haskell compiler's type checker. In this way a partial correctness proof is given for these manipulations.

4.3.1 Abstract Syntax Trees

Before presenting the ideas behind typed abstract syntax trees we consider a simple language consisting of integer constants, equality operator, addition, and if-expressions. The untyped abstract syntax of this language can be represented by the following data type:

```
data Expr = Int    Int
          | Equals Expr Expr
          | Add    Expr Expr
          | If     Expr Expr Expr
```

The expression **if** $3 \equiv 5$ **then** 1 **else** $4 + 5$ is encoded as:

If (*Equals* (*Int* 3) (*Int* 5)) (*Int* 1) (*Add* (*Int* 4) (*Int* 5))

The static semantics of this language are defined by the typing judgments in Figure 4.1. The data type as given here can represent all syntactically correct terms. For example leaving out the **else**-branch of an **if**-statement leads to a type-error. However, the type does not prevent us from constructing terms that violate the typing judgments: the non-typeable expression **if** 3 **then** 4 **else** 5 is happily represented as *If* (*Int* 3) (*Int* 4) (*Int* 5).

4.3.2 Phantom Types

Using phantom types [39] to prevent the construction of ill-typed terms is a well-known technique. The data type *Expr* is labeled with an type parameter.

A value of type *Expr t* represents an object language terms of type t. The type parameter is only used to hold a type, it is never physically present, hence the name phantom type.

```
data Expr a = Int Int
            | Equals (Expr Int)  (Expr Int)
            | Add    (Expr Int)  (Expr Int)
            | If     (Expr Bool) (Expr a) (Expr a)
```

We define a number of smart constructors. The type signatures are important, since they constrain the type of the constructed term. Without the explicit type signature the type assigned to *int* would be the more general $(Int \to Expr\ a)$, which is undesirable.

```
int    :: Int → Expr Int
int    = Int
equals :: Expr Int → Expr Int → Expr Bool
equals = Equals
add    :: Expr Int → Expr Int → Expr Int
add    = Add
```

Note the correspondence between the types of the smart constructors and the typing judgments. Note also that we do not need a constructor for the **if** since in that alternative the type t is not constrained to some specific type. When a programmer uses the smart constructors, instead of the normal constructors, we are sure that he only constructs well-typed terms. This could be enforced by simply hiding the constructors *Int*, *Equals*, and *Add*. For example: the expression **if** $3 \equiv 5$ **then** 1 **else** $4 + 5$ is now easily encoded as: *If* (*equals* (*int* 3) (*int* 5)) (*int* 1) (*add* (*int* 4) (*int* 5)), and trying to represent the incorrect expression **if** 3 **then** 4 **else** 5 will lead to a type error from the Haskell compiler.

We would like to define an interpreter for *Expr* as follows:

```
eval :: Expr a → a
eval (Int x)      = x
eval (Equals x y) = eval x ≡ eval y
eval (Add x y)    = eval x + eval y
eval (If c t e)   = if eval c then eval t else eval e
```

Unfortunately, this function is rejected by the Haskell type checker, because the result types of the cases do not match. On the other hand, the function *eval* is not really bad, when we pass it only well-typed terms nothing will go wrong. For example, if a well-typed term of type $Expr\ \alpha$ matches the *Add* constructor, then we know that the type α is actually *Int*. The phantom types technique ensures that we can only construct well-typed terms, but we cannot take advantage of this knowledge when deconstructing a term using pattern matching.

We now present a new data type that solves this problem. The data type *Expr* is labeled with a type variable. A value of type *Expr t* represents an object

language terms of type t. Equality types ([5]) are used to express the constraints imposed by the typing judgments. A value of the equality type $(Equal\ a\ b)$ can be seen an a proof that the types a and b are equal, and can thus be used to restrict the type variable t in the definition of *Expr*:

```
data Expr t
  = Int    (Equal Int t)   Int
  | Equals (Equal Bool t) (Expr Int) (Expr Int)
  | Add    (Equal Int t)   (Expr Int) (Expr Int)
  | If     (Expr Bool)     (Expr t)   (Expr t)
```

For the constructors *Int* and *Add* the type variable t is restricted to the type *Int*, and for the constructor *Equals* it is restricted to *Bool*. We define a number of smart constructors that simply apply the corresponding constructors to $self :: Equal\ a\ a$, the only non-bottom value of the equality type.

```
int    :: Int → Expr Int
int    = Int self
equals :: Expr Int → Expr Int → Expr Bool
equals = Equals self
add    :: Expr Int → Expr Int → Expr Int
add    = Add self
```

Note that the type signatures do not restrict the types of the smart constructors as in the phantom types approach. They could be left out, and it is also not necessary to hide the actual constructors of the type *Expr*.

Using the function $cast :: Equal\ a\ b \to (a \to b)$, which is defined in the next section, we can easily define the *eval* function for *Expr* as follows:

```
eval :: Expr a → a
eval (Int eq x)       = cast eq x
eval (Equals eq l r) = cast eq (eval l ≡ eval r)
eval (Add eq l r)     = cast eq (eval l + eval r)
eval (If c t e)       = if eval c then eval t else eval e
```

4.3.3 The Equality Type

We saw how equality types ([5]) played a crucial role in the definition of a fully typed *Expr*. In this section we discuss how they are defined and used. One can view a language such as Haskell from the perspective of the Curry-Howard isomorphism: types correspond to logical propositions; a non-bottom program with a particular type is a proof of the corresponding proposition.

The equality between types is encoded as a type constructor $(Equal\ a\ b)$. A value of type $p :: Equal\ a\ b$ can be seen as a proof of the proposition that a equals b.

```
data Equal a b = Eq (∀φ . φ a → φ b)
self           :: Equal a a
```

$$
\begin{array}{ll}
\mathit{self} & = \mathit{Eq}\ \mathit{id} \\
\mathit{castF} & :: \mathit{Equal}\ a\ b \to \varphi\ a \to \varphi\ b \\
\mathit{castF}\ (\mathit{Eq}\ f) & = f
\end{array}
$$

The intuition behind this definition (also known as Leibnitz' equality) is the following: two types are equal if, and only if they are interchangeable in any context. This context is represented by any Haskell type constructor φ. The value $\mathit{self} :: \mathit{Equal}\ a\ a$ is the only non-bottom inhabitant of the *Equal* type.

From a proof of type $(\mathit{Equal}\ a\ b)$, a casting function of type $\varphi\ a \to \varphi\ b$, for any context φ can be extracted. For example we can construct functions $\mathit{cast} :: \mathit{Equal}\ a\ b \to (a \to b)$, and $\mathit{invcast} :: \mathit{Equal}\ a\ b \to (b \to a)$ that allow us to cast between a and b.

Figure 4.2 shows the definition of the *Equal* type, the casting operators and combinators that can be used to build larger equality proofs from smaller ones. The implementations of the casting operators and combinators can be found in Appendix 4.9. Finally in Figure 4.2 we give the axiom *pairParts*, which states that if two pairs are equals, we can extract proofs that the parts of the pairs are equal. According to Hinze [12] this function cannot be constructed because the law $\mathit{id} :: (a, b) \to (a, b) = \mathit{lift2}\ (,)\ \mathit{id}\ \mathit{id}$ does not have an extensional counterpart in Haskell, and so we take it as an axiom.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Equal}\ a\ b = \mathit{Eq}\ (\forall f\ .\ f\ a \to f\ b) \\
\quad \text{-- Reflexivity} \\
\mathit{self} \quad :: \mathit{Equal}\ a\ a \\
\quad \text{-- Symmetry} \\
\mathit{inv} \quad :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ a \\
\quad \text{-- Transitivity} \\
\mathit{trans} \quad :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ c \to \mathit{Equal}\ a\ c \\
\quad \text{-- Congruence} \\
\mathit{subF} \quad :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ (t\ a)\ (t\ b) \\
\mathit{subF2} \quad :: \mathit{Equal}\ a\ c \to \mathit{Equal}\ b\ d \to \mathit{Equal}\ (t\ a\ b)\ (t\ c\ d) \\
\quad \text{-- Casting operators} \\
\mathit{cast} \quad :: \mathit{Equal}\ a\ b \to a \to b \\
\mathit{invcast} \quad :: \mathit{Equal}\ b\ a \to b \to a \\
\mathit{castF} \quad :: \mathit{Equal}\ a\ b \to t\ a \to t\ b \\
\quad \text{-- Axioms} \\
\mathit{pairParts} :: \mathit{Equal}\ (a, b)\ (c, d) \to (\mathit{Equal}\ a\ c, \mathit{Equal}\ b\ d)
\end{array}
$$

Figure 4.2: *Equal* type and operators

4.3.4 References

In the paper by Pasalic [45], scope-rules of object languages are implemented in terms of an explicit encoding of references. The implementation can deal with nested scopes and pattern bindings. The most important feature however is that references can be observed and compared for equality.

This same feature enables us to observe the nonterminals in the right-hand side of a production rule, and allows us to detect cycles.

We will label environment values with a type that represents the types of the values contained in the environment; for this we use nested pairs of types. The encoding of references of type *Ref env a*, which represents a pointer to a value of type *a* in an environment labeled with type *env*, is inspired by de Bruijn indices:

```
data Ref env a
  = ∃  env' . Zero (Equal env (a, env'))
  | ∃x env' . Suc  (Equal env (x, env')) (Ref env' a)
```

As before we define smart constructors that pass the equality proof *self* to the corresponding constructor functions:

```
zero :: Ref (a, env) a
zero = Zero self
suc  :: Ref env' a → Ref (b, env') a
suc  = Suc self
```

The number of *Suc*-nodes in a reference determines to which value in the environment a reference points.

```
deref :: Ref env a → (env → a)
deref (Zero eq)     = fst . cast eq
deref (Suc eq ref) = deref ref . snd . cast eq
```

Two arbitrary references can be compared for equality as long as they point into environments that are labeled with the same sequence of types. If the comparison succeeds a proof that the types of the values they point to are equal is returned.

```
equalRef  :: Ref env a → Ref env b → Maybe (Equal a b)
equalRef (Zero eq1) (Zero eq2)
  =      let (eq, _) = pairParts (inv eq1 `trans` eq2)
         in Just eq
equalRef (Suc eq1 ref1) (Suc eq2 ref2)
  =      let (_, eq) = pairParts (inv eq1 `trans` eq2)
         in equalRef (cast (subF2 eq self) ref1) ref2
equalRef _ _ = Nothing
```

A drawback of implementing an environment as nested pairs is that the time needed for a variable lookup is linear in the size of the environment. This is

undesirable, and can be avoided. When 'compiling' an object-language term we can take advantage of the fact that the environment is partially static. The shape of the environment only depends on the structure of the term being interpreted. Pasalic et al. introduce a partially static environment (*Env*) that takes advantage of this fact, speeding up their interpreters.

```
data Env f env
       =            EMPTY
       | ∃a env' . EXT (Equal env (a, env'))
                       (f a) (Env f env')
  -- smart constructors
empty :: Env f ()
empty = EMPTY
ext   :: f a → Env f env → Env f (a, env)
ext   = EXT self
```

The type *Env* is parametrized with a type constructor f, making it slightly more general than the original definition. The dereferencing operator is defined in a similar way as before:

```
derefEnv :: Ref env a → Env f env → f a
derefEnv (Zero eq1) (EXT eq2 res _)
   = let (eq3, _) = pairParts (inv eq2 `trans` eq1)
     in castF eq3 res
derefEnv (Suc eq1 ref) (EXT eq2 _ rest)
   = let (_, eq3) = pairParts (inv eq2 `trans` eq1)
     in derefEnv ref (castF eq3 rest)
derefEnv _ EMPTY = error "environment is empty"
```

Apart from speeding up interpreters, the *Env* type has another advantage, not noticed by Pasalic. The environment can store object-language terms containing references that point to other terms in the same environment. When using nested pairs this would lead to infinite types. It is this property that enables us to analyze the parsers for left-recursion etc. How to do this will be the subject of the next section.

4.4 Parsing combinators

Embedding a domain-specific language by defining a library of combinators is common practice in Haskell. There are many parsing combinator libraries around that enable a programmer to define parsers that closely resemble EBNF notation. We briefly present the interface of a parsing combinator library in figure 4.3. The parser *symbol* accepts solely the given character as input. If this character is encountered, *symbol* consumes and returns this character, otherwise it fails. The parser *succeed* does not consume any input always succeeds with the given input, whereas the *failp* always fails. The operator ≪ denotes sequential

```
infixl 4 <*>
infixr 3 <|>
type Parser a
symbol  :: Char → Parser Char
succeed :: a → Parser a
(<*>)   :: Parser (a → b) → Parser a → Parser b
(<|>)   :: Parser a → Parser a → Parser a
failp   :: Parser a
parse   :: Parser a → a
```

Figure 4.3: Parser combinators

```
type Parser a = [Char] → [(a, [Char])]
p <|> q       = λinp → p inp ++ q inp
symbol x      = λinp → case c : cs | x ≡ c → [(c, cs)]
                                   _       → []
p <*> q       = λinp → [ (f x, inp2)
                       | (f, inp1) ← p inp
                       , (x, inp2) ← q inp1
                       ]
failp         = λinp → []
succeed x     = λinp → [(x, inp)]
parse p input = case p input of
                  [(res, "")] → res
                  _           → error "parse error"
```

Figure 4.4: List-of-successes implementation

composition of two parsers, where the result of the first parser is applied to the result of the second. The operator <|> expresses a choice between two parsers.

For the sake of completeness we give a simple list-of-successes [64] implementation of this parser interface in figure 4.4. For the rest of this paper the actual implementation however does not matter.

Many useful combinators can be built on top of these basic combinators. A small selection that we use in this paper is presented in Figure 4.5.

4.5 Analyzing Grammars

4.5.1 Representing grammars

The idea of combinators that analyze the grammar they describe, before constructing the real parser, is introduced in the fast error-repairing combinators

```
infix 5 <$>
(<$>)      :: (a → b) → Parser a → Parser b
f <$> p    = succeed f <*> p
many       :: Parser a → Parser [a]
many       = let ps = many p
             in (:) <$> p <*> ps <|> succeed []
many1      :: Parser a → Parser [a]
many1 p    = (:) <$> p <*> many p
choice     :: [Parser a] → Parser a
choice ps  = foldr (<|>) failp ps
```

Figure 4.5: Parser combinators

of Swierstra and Duponcheel ([58]). The analysis includes determining which productions may derive the empty string, the computation of *firsts* sets and the construction of fast lookup tables. Now we take the idea of self-analyzing combinators a step further, and determine whether a grammar is left-recursive. If so we we transform it into an equivalent, non-left-recursive one. The transformed grammar can subsequently be used to construct an efficient top-down parser, based on the well-known techniques.

One of the reasons that this has not been done before is that with the combinators given we only have an implicit representation of the parsers available; we do not have explicit access to the call graph of the parsers, and thus cannot detect left-recursion, let alone to do something about it. This is also the reason that it was thus far impossible to check statically whether the represented grammar was indeed *LL(1)*; in order to do so one needs access to all the calling points of a parser –in order to be able to compute its set of *followers*– and this is not something that can be done, unless one resorts to approaches such as template meta-programming [54] for constructing parsers.

What we are after is thus a more explicit representation for grammars and their production rules, so they can be inspected and modified, before generating real parsers, while at the same time keeping the elegance of notation and the possibility for abstraction provided by the combinator based approach.

We start by representing a grammar as an environment that contains for each non-terminal we want to refer to a description of its combined (using *Choice*) right hand sides.

```
type Grammar env = Env (Parser env) env
```

The occurrence of a nonterminal in the right-hand side of a production is represented by a reference to the corresponding component in the grammar representation. The data type *Parser* defines the abstract syntax trees of the right-hand sides of production rules. Note that this *Parser* type is also labeled

with the types returned by the parsers contained in the grammar, thus enforcing that references are guaranteed to refer to parsers that actually form part of the grammar.

```
data Parser env a
  =       Symbol  (Equal Char a)          Char
  |       Succeed a
  |       Fail
  |       Choice  (Parser env a)          (Parser env a)
  | ∃b . Seq     (Parser env (b → a)) (Parser env b)
  | ∃x . Many    (Equal [x] a)           (Parser env x)
  |       NT      (Ref env a)
```

Note that the constructor *Many* that represents the zero-or-more occurrences operator is not strictly necessary. It will however prove to be very convenient in the left-recursion removal.

The combinator library interface can again be implemented by a number of smart constructors:

```
nt      :: Ref env a → Parser env a
nt ref  = NT ref
symbol  :: Char → Parser env Char
symbol  = Symbol self
succeed :: a → Parser env a
succeed = Succeed
failp   :: Parser env a
failp   = Fail


(<|>)     :: Parser env a → Parser env a → Parser env a
p <|> Fail = p
Fail <|> q = q
p <|> q    = Choice p q
(<*>)     :: Parser env (a → b) → Parser env a → Parser env b
p <*> q    = Seq p q
many      :: Parser env a → Parser env [a]
many      = Many self
```

4.5.2 Compiling grammars

Using the smart constructors just introduced we now can produce an explicit representation of a grammar, that can be inspected and transformed and used as a starting point for generating real parsers.

We use qualified names to avoid clashes with the names of the types and operators of a real parser combinator library(called *PL*). The function *compile*, which maps the description of a single parser to a real parser takes as its first

argument an environment containing compiled parsers, and as second argument the abstract syntax tree of the combined production rules of one nonterminal. The function interprets the abstract syntax tree, using the environment to convert references to real parsers, and yields a compiled parser. An *NT* constructor is interpreted by looking up the corresponding parser in the environment. All other constructors are interpreted by calling its associated combinator from the real parser library.

```
compile :: Env PL.Parser env
  →      Parser env a
  →      PL.Parser a
compile parsers prod =
  case prod of
    NT ref      → derefEnv ref parsers
    Choice p q  → comp pPL. <|> comp q
    Symbol eq c → castF eq (PL.symbol c)
    Succeed x   → PL.succeed x
    Seq p q     → comp pPL. <*> comp q
    Many eq p   → castF eq (PL.many (comp p))
    Fail        → PL.failp
  where comp = compile parsers
```

Compiling an entire grammar proceeds as follows. Firstly *mapEnv* applies the function *compile* to every production in the grammar, converting an environment containing parser descriptions into an environment containing parsers, where the function compile gets passed the final result to lookup the references.

```
compileGrammar :: Grammar env → Env PL.Parser env
compileGrammar gram =
  let parsers = mapEnv (compile parsers) gram
  in  parsers
mapEnv :: (∀a . f a → g a) → Env f env → Env g env
mapEnv f EMPTY          = EMPTY
mapEnv f (EXT eq x rest) = EXT eq (f x) (mapEnv f rest)
```

The function *compileGrammar* still only returns working parsers for grammars that are not left-recursive. This leaves us with two problems: first of all we need some functions to help constructing a *Grammar*, in such a way that the corresponding code still looks like a grammar. Secondly we need a function that transforms a left-recursive grammar into an equivalent non-left-recursive one.

4.5.3 Removing left-recursion

The left-recursion removal algorithm proceeds as follows. For each nonterminal the corresponding production is split into three parts:

- an empty part

- non-left-recursive alternatives
- left-recursive alternatives

The empty part is a *Maybe* value, if the production can derive the empty string it contains the semantics of the empty derivation, otherwise it is *Nothing*. The non-left-recursive alternatives are represented as a parser that is not left-recursive and does not derive the empty string. The left-recursive part is also represented as a parser. The left-recursive call to the nonterminal is already stripped off. Therefore this parser yields a function, that needs the semantics of the left-recursive nonterminal as argument. If the left-recursive part is *Fail*, then the production has no left-recursive alternatives, and does not need to be transformed. Otherwise the transformed production is constructed by combining the three parts using the function *transform*, yielding a production that is no longer left-recursive.

```
transform :: Maybe a → Parser e a → Parser e (a → a) → Parser e a
transform empty nonlefts lefts =
  let glue = foldl (flip ($))
  in  glue <$> (maybe failp succeed empty <|> nonlefts) <*> many lefts
```

The function *removeLeft* iterates over the nonterminals of the grammar (represented as a *Env* containing references). For each nonterminal it determines whether its production is left-recursive, and if so replaces this production by a transformed version. When all nonterminals are checked the resulting grammar is no longer left-recursive.

```
removeLeft :: Env (Ref env) x → Grammar env → Grammar env
removeLeft EMPTY env = env
removeLeft (EXT eq ref rest) env
   = let (e, n, l) = unfold env ref
         env'    | isFail l          = env
                 | otherwise         = writeEnv ref (transform e n l) env
     in  removeLeft rest env'
isFail Fail = True
isFail _ = False
```

The function *writeEnv* takes a reference and a value and stores the value at the position indicated by the reference.

```
writeEnv :: Ref env a → f a → Env f env → Env f env
writeEnv _ _ EMPTY = error "environment is empty"
writeEnv (Zero eq1) x (EXT eq2 _ rest)
   = let (eq3, _) = pairParts (inv eq1 `trans` eq2)
     in  EXT eq2 (castF eq3 x) rest
writeEnv (Suc eq1 ref) x (EXT eq2 fa rest)
   = let (eq4, eq3) = pairParts (inv eq2 `trans` eq1)
     in  EXT eq1 (castF eq4 fa) (writeEnv ref x (castF eq3 rest))
```

The function *unfold* does the actual analysis. It takes the grammar and the nonterminal to be analyzed as arguments and splits the production corresponding to the nonterminal into three components.

$$\begin{aligned} &\mathit{unfold} :: \mathit{Grammar}\ \mathit{env} \\ &\qquad \to \mathit{Ref}\ \mathit{env}\ \mathit{nt} \\ &\qquad \to (\mathit{Maybe}\ \mathit{nt}, \mathit{Parser}\ \mathit{env}\ \mathit{nt}, \mathit{Parser}\ \mathit{env}\ (\mathit{nt} \to \mathit{nt})) \end{aligned}$$

The implementation of *unfold* can be found in Appendix 4.10. There are two things to be noted here. In the first place the whole transformation is a well-typed Haskell program, and as such we have given an implicit partial correctness proof of our transformation; depending on your point of view this can be seen as an advantage of this approach or as an additional burden for the programmer of this transformation library.

The second observation is that we have taken a quite straightforward approach here, but there is nothing that prevents one from taking smarter approaches to the transformation described here or performing other kinds of transformations at the same time, like left-factorization etc.

4.6 Constructing grammars

When writing grammars we should no longer use the normal Haskell way of making bindings, but use explicit references instead. This makes the direct construction of *Grammar*s a tedious job.

Consider for example the following grammar:

$$\begin{aligned} P &\to Q\ \texttt{"a"} \\ Q &\to P\ \texttt{"b"} \mid \texttt{"c"} \end{aligned}$$

The nonterminals P, and Q are represented by the references *zero* and *suc zero* respectively, leading to the following implementation for the grammar.

$$\begin{aligned} &\textbf{infixr}\ 1\ `\mathit{ext}` \\ &\mathit{example} :: \mathit{Grammar}\ (\mathit{String}, (\mathit{String}, ())) \\ &\mathit{example} = \\ &\quad (+\!\!+) \mathbin{<\$>} \mathit{nt}\ (\mathit{suc}\ \mathit{zero}) \mathbin{<\!\!*\!\!>} \mathit{token}\ \texttt{"a"} \\ &\qquad `\mathit{ext}` \\ &\quad (+\!\!+) \mathbin{<\$>} \mathit{nt}\ \mathit{zero} \mathbin{<\!\!*\!\!>} \mathit{token}\ \texttt{"b"} \mathbin{<|>} \mathit{token}\ \texttt{"c"} \\ &\qquad `\mathit{ext}` \\ &\quad \mathit{empty} \\ &\mathit{token}\ \texttt{""} \quad\ = \mathit{succeed}\ \texttt{""} \\ &\mathit{token}\ (c : cs) = (:) \mathbin{<\$>} \mathit{symbol}\ c \mathbin{<\!\!*\!\!>} \mathit{token}\ cs \end{aligned}$$

The definition above is quite unreadable, and things get much worse when dealing with larger grammars. To improve the situation we take inspiration from the way mutually recursive **let** and **mdo** expressions [19, 46] are translated into

an application of a fix-point operator to a lambda term that takes the identifiers from the declarations as argument and returns a product containing the bodies of the declarations as result.

```
infixr 1 `andalso`
example :: Grammar (String, (String, ()))
example =
  fixRefs
  (λ~(p, (q, _)) →
    (++) <$> q <*> token "a"
      `andalso`
    (++) <$> p <*> token "b" <|> token "c"
      `andalso`
    done
  )
```

The argument of the lambda term is a nested product containing nonterminal references, which are given a name by pattern matching. The lazy pattern match is very important, because a fix-point operator is applied to the lambda term. The body of the lambda term consists of a number of parsers separated by *andalso*. The sequence is terminated by *done*. The first element in the nested product is a nonterminal reference that points to the first parser, the second reference points to the second parser, and so on. Hence the number of identifiers in the pattern should be the same as the number of parsers in the body of the lambda term. The body of the lambda collects the parser in a *Env*, and computes the reference for each parser. These references are passed to the lambda term by the fix-point operator *fixRefs*.

The function *andalso* is implemented as follows. It takes a parser and a tuple containing two *Env*'s and an '*unpack*' function. The first *Env* is the grammar constructed thus-far and contains a number of parsers. The second contains references to these parsers and represents the nonterminals of the grammar. The *unpack* function can be used to extract all references and convert them in a nested product. The function *andalso* adds the parser p to the grammar. Furthermore it increments all references by applying the function *suc* to them and adds a new reference. Finally the *unpack* function is updated, so it can also extract the newly added reference.

```
andalso p (env, refs, unpack)
  = (ext p env, ext zero (mapEnv suc refs), unpack')
  where unpack' (EXT eq fa rest)
              = let (eq1, eq2) = pairParts eq
                in  (nt (castF (inv eq1) fa)
                    , unpack (castF (inv eq2) rest)
                    )
```

The function *done* simply initializes the grammar, sequence of nonterminals and the unpack function:

$$done = (empty, empty, \lambda\_ \rightarrow ())$$

Finally the function *fixRefs* is defined as follows:

$$\begin{aligned} fixRefs\ f = \ & \textbf{let}\ (grammar, refs, unpack) = f\ (unpack\ refs) \\ & \textbf{in}\ removeLeft\ refs\ grammar \end{aligned}$$

The fix-point operator takes a function f as argument. This function returns a grammar, a sequence of references and an *unpack* function. The *unpack* function is applied to the sequences of references and yields a nested product that is passed to the function f. Finally *removeLeft* is applied to the the grammar resulting in a grammar without left-recursion.

A working parser for the simple example language, assuming that P is the start symbol, can be obtained as follows:

$$\begin{aligned} & parseP :: PL.Parser\ String \\ & parseP = derefEnv\ zero\ (compileGrammar\ (example)) \end{aligned}$$

4.7 Syntactic extensions

The notational elegance of normal parser combinators is lost due to the use of a custom fix-point operator to make recursion explicit. At first sight one might think that the **mdo** notation [19] could help us here, since the meaning of recursive **mdo** bindings is defined through a programmer defined fix-point operator. If we could define our grammars as a state-monad recording an environment containing the parsers of the grammar, our example grammar could be written as follows:

$$\begin{aligned} example = \textbf{mdo}\ & p \leftarrow (+\!\!+) \mathbin{<\$>} q \mathbin{<\!\!*\!\!>} token\ \texttt{"a"} \\ & q \leftarrow (+\!\!+) \mathbin{<\$>} p \mathbin{<\!\!*\!\!>} token\ \texttt{"b"} \mathbin{<|>} token\ \texttt{"c"} \\ & \ldots \end{aligned}$$

Unfortunately the **mdo** syntax cannot be used: our types cannot be made an instance of the *Monad* class because the type of the would-be monad changes whenever we extend the environment with a new parser.

We could also add new syntactic sugar as is done for the arrow library [30]. Programming directly at the arrow combinator level is very tedious, however using Paterson's arrow notation [44] makes programming with arrows entirely practical. Hard-wiring a special notation for combinator parsers into a Haskell compiler as is done for **mdo** and arrows would not make sense. If everyone proceeds along this path, the compiler would soon be cluttered with extensions for all kinds of domain specific languages. We advocate the syntax macro approach [4, 37, 10]. Syntax Macros provide a generic mechanism for extending a compiler with syntactic sugar. A programmer can load the desired language extensions together with the combinator library. The **mdo** and arrow notations could be

implemented as syntax macros, as well as many of the translation schemes in the Haskell Report [46] (**if-then-else**, **do** notation, etc.).

At the moment, we are working on a prototype implementation of syntax macros [4] for GHC [21]. As an example of the use of syntax macro we show how a more elegant notation for defining grammars can be specified. A better notation for the example grammar could be:

$$
\begin{array}{l}
\mathit{example} = \textbf{grammar} \\
\quad p \leftarrow (+\!\!+) \mathbin{<\$>} q \mathbin{<\!\!*\!\!>} \mathit{token}\ \texttt{"a"}; \\
\quad q \leftarrow (+\!\!+) \mathbin{<\$>} p \mathbin{<\!\!*\!\!>} \mathit{token}\ \texttt{"b"} \\
\qquad\qquad \mathbin{<|>} \mathit{token}\ \texttt{"c"};
\end{array}
$$

A grammar is simply denoted by the keyword **grammar** followed by a number of bindings (the production rules of the grammar).

This simple syntactic extension is implemented as a macro as follows. Firstly we need to declare the nonterminals that are used in the macro. The nonterminals `varid`, `exp`, and `exp10` are existing nonterminals of the Haskell grammar as defined in the Haskell Report. They represent lowercase identifiers, expressions, and expressions at the precedence level of **let**, respectively. The nonterminal `prods` is a new nonterminal and represents the production rules of a grammar expression. The type of the syntax trees derived by this nonterminal is a pair of a pattern containing the variables in the production rules and and expression consisting of the parser of the productions.

```
nonterminals:
varid  :: String
exp    :: Exp
exp10  :: Exp
prods  :: (Pat, Exp)
```

The new notation of grammar expressions is defined by the following macro rules:

```
rules:
exp10 ::= "grammar" (ids,ps)=prods
      =>  [| fixRefs (\ ~ $ids -> $ps ) |]

prods ::= => (WildP, [|done|])
prods ::= v=varid "<-" e=exp ";" (ids,ps)=prods
      =>  let var = VarP v
          in  ( [|p ($var, $ids) |]
              , [|  $e `andalso` $ps|]
              )
```

The syntax of the syntax macros is inspired by Cardelli et al. [10] and Template Haskell [54]. Each macro starts with a BNF-like production rule at the left-hand side of the `=>` symbol, that defines the new concrete syntax. The

right-hand side defines how the new syntax is mapped on the abstract syntax of Haskell. The abstract syntax can be denoted using Template Haskell's notation.

The first macro states that the parser for `exp10` is extended with a new alternative that recognizes the symbol **grammar** followed by a number of production rules (`prods`). The result of `prods` is a pair (ids, ps) consisting of the syntax trees for a pattern and an expression. The resulting abstract syntax tree defined at the right-hand side of `=>` is an application of *fixRefs* to a lambda term that matches the pattern with identifiers and returns the expression containing the parsers.

The macros for `prods` define the syntax for a sequence of productions. The first line defines the semantics for the empty sequence, for which a pair is returned containing a wild card pattern, and the function *done*. The second line says that a non-empty sequence of productions consists of an identifier, an $\leftarrow$ symbol, and an expression followed by a sequence of other productions. The identifier is converted into a pattern identifier and combined with the other identifiers. The expression is combined with the other expressions using the function *andalso*.

More macros can be added to make the notation of parsing combinators look like that of a parser generator such as Happy [22]. Then we can write our parsers using an elegant notation, and at the same time benefit from Haskell's abstraction mechanism and type system. Syntax macros are applicable to many other domain specific languages. We think a syntax macro mechanism makes Haskell an even better tool for developing EDSL's.

4.8 Conclusions

We have shown how, by using an environment that is labeled by a type constructor and a cartesian product of types, we can represent programming structures that contain internal references, and how to inspect such internal references. In order to do so we had to invent a data type that described the top level of our embedded language, up-to the point that all references were visible. Beneath that level ordinary Haskell expressions can be used. This works especially well in the case of combinator based parsers, since there the top level structure contains elementary parsers and combinators that construct parsers, whereas in the semantic functions that we use in such parsers no further references to parsers do occur.

Once we have this data structure in our hands we can analyze and transform it. We have shown how we can transform a left-recursive grammar into a non-left-recursive grammar. By using more advanced techniques one may try to have the resulting grammars as small as possible, but we have abstained here from such optimizations since they are outside the scope of this paper. Although we did not do so, we think it is even possible to generate bottom-up parsers out of such descriptions.

We want to emphasize once more that this technique not only applies to combinator parsers; all situations in which one is interested in some form of a

typed call graph can be dealt with in this way.

The work on observable sharing by Claessen and Sands [15] also addresses the problem of detecting cycles. Their comparison operator ($\Leftrightarrow :: \mathit{Ref}\ a \rightarrow \mathit{Ref}\ a \rightarrow \mathit{Bool}$) can only compare references that are labeled with the same type. This approach works fine if the objects in which we want to detect cycles all have the same type. However, this is not the case for combinator parsers. Our comparison operator: $\mathit{equalRef} :: \mathit{Ref}\ \mathit{env}\ a \rightarrow \mathit{Ref}\ \mathit{env}\ b \rightarrow \mathit{Maybe}\ (\mathit{Equal}\ a\ b)$, allows us to compare arbitrary references, provided that they point into the same environment.

The work described here is closely related to work on meta-programming using typed abstract syntax by Pasalic [45]. In trying to solve the left-recursion removal problem we ended up with the same data type as Pasalic in [45], but for a different purpose. Where Pasalic is using the partially static environments to obtain more efficient interpreters, we use it as an indirection to the structure we are actually constructing, thus circumventing the construction of an infinite data type: the references do not directly refer to the environment being constructed, but are labeled with a type describing the sequence of types associated with the values in such an environment, which is represented by a similar label to this environment. Later, when following the encoded indirections, the Haskell type system enforces that the kind of environment they were supposed to index into corresponds to the type of the environment that is actually used.

After having gone through all the code one might wonder why things are getting so complicated. Weren't combinator libraries supposed to be nice, small and elegant? Our conclusion is that if one wants to write interesting, semantically rich embedded languages one cannot stay away from the work that is usually done by separate compilers and program generators, such as type checking, abstract interpretation, code transformation, the construction of new programs etc. Trying to achieve the same effect in an embedded way, i.e. in the context of a fully typed formalism like Haskell brings the extra burden of making sure that the "compilers" are type correct. Doing so however also has its advantages: getting the type of the analyzing and transforming program correct gives us a partial correctness proof of the analysis and transformation for free ([65]. Using template meta-programming it may be easier to get something working, but may require a separate proof effort to convince oneself that the resulting code is not only type correct, but also semantically correct. Let us finish by paraphrasing one of our students who said "I do not like Haskell because the compiler is always complaining about my program.", by saying "We like Haskell because the compiler always complains about our incorrect programs.".

4.9 Equality types

```
data Equal a b = Eq (∀f . f a → f b)
  -- Reflexivity
self :: Equal a a
self = Eq id


  -- Symmetry
inv    :: Equal a b → Equal b a
inv eq = unFlip . castF eq . Flip $ self
data Flip a b = Flip {unFlip :: Equal b a}


  -- Transitivity
trans :: Equal a b → Equal b c → Equal a c
trans (Eq cast) (Eq b2c) = Eq (b2c . cast)


  -- Congruence
subF :: Equal a b → Equal (t a) (t b)
subF (Eq ab) = case ab (H self) of
                 H x → x
subF2 :: Equal a c → Equal b d → Equal (t a b) (t c d)
subF2 (Eq ac) (Eq bd) =
      case ac (F self) of
        F x → case bd (G x) of
          G y → y
data F f a c x   = F (Equal (f a c) (f x c))
data G f a c b x = G (Equal (f a c) (f b x))
data H f a x     = H (Equal (f a) (f x))


  -- Casting operators
cast          :: Equal a b → a → b
cast eq       = castF eq id
invcast       :: Equal a b → b → a
invcast eq    = cast (inv eq)
castF         :: Equal a b → t a → t b
castF (Eq f) = f
  -- Axiom
pairParts   :: Equal (a, b) (c, d) → (Equal a c, Equal b d)
pairParts _ = (unsafeCoerce self, unsafeCoerce self)
```

4.10 Definition of *unfold*

```
unfold  :: Grammar env → Ref env nt
        → (Maybe nt, Parser env nt, Parser env (nt → nt))
unfold env (nt :: Ref env nt) = unfold' [] (derefEnv nt env) where
  unfold' :: [Exists (Ref env)] → Parser env a
          → (Maybe a, Parser env a, Parser env (nt → a))
  unfold' visited p@(NT ref) =
    case equalRef nt ref of
      Just eq                          → (Nothing, Fail, succeed (cast eq))
      Nothing | visited `contains` ref → (Nothing, NT ref, Fail)
              | otherwise              → unfold' (Exists ref : visited)
                                                 (derefEnv ref env)
  unfold' _ p@(Symbol eq x)  = (Nothing, p, Fail)
  unfold' _ (Succeed x)      = (Just x, Fail, Fail)
  unfold' _ Fail             = (Nothing, Fail, Fail)
  unfold' visited (Many eq p) =
    let (pe, pn, pl) = unfold' visited p
        nlefts' = castF eq (many1 pn)
        nlefts  = if isFail lefts then castF eq (many1 p) else nlefts'
        lefts   = (λf xs nt → cast eq (f nt : xs)) <$> pl <*> many p
    in  (castF eq (Just []), nlefts, lefts)
  unfold' visited (Choice p q) = let (pe, pn, pl) = unfold' visited p
                                     (qe, qn, ql) = unfold' visited q
                                 in  (pe `xor` qe, pn <|> qn, pl <|> ql)


unfold' visited r@(Seq p q) =
  let (pe, pn, pl) = unfold' visited p
      (qe, qn, ql) = unfold' visited q
      empty        = bothEmpty pe qe
      nlefts'      = let lnl = pn <*> q
                         rnl = case pe of
                                 Nothing → Fail
                                 Just f  → succeed f <*> qn
                     in  lnl <|> rnl
      nlefts       = if isFail lefts ∧ isNothing empty then r else nlefts'
      lefts        = let llr = succeed flip <*> pl <*> q
                         rlr = case pe of
                                 Nothing → Fail
                                 Just f  → succeed (f .) <*> ql
                     in  (llr <|> rlr)
  in (empty, nlefts, lefts)


contains refs ref = any (λ(Exists x) → isJust . equalRef ref $ x) refs
```

```
xor :: Maybe a → Maybe a → Maybe a
xor (Just _) (Just _) = error "ambiguous"
xor Nothing r         = r
xor l       Nothing   = l
```

```
bothEmpty :: Maybe (a → b) → Maybe a → Maybe b
bothEmpty (Just f) (Just x) = Just (f x)
bothEmpty _        _        = Nothing
```

```
data Exists f = ∃x . Exists (f x)
```

Chapter 5

Typed Transformations of Typed Abstract Syntax

This is a slightly edited version of a paper originally published as:
Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

Typed Transformations of Typed Abstract Syntax

Arthur I. Baars
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Valencia, Spain

S. Doaitse Swierstra
Department of Computer Science
Utrecht University
Utrecht, The Netherlands

Marcos Viera
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Abstract: Advantages of embedded domain-specific languages are that one does not have to implement a separate type system nor an abstraction mechanism, since these are directly borrowed from the host language. Straightforward implementations of embedded domain-specific languages (EDSL) map the semantics of the embedded language onto a function in the host language. The semantic mappings are usually compositional, i.e. they directly follow the syntax of the embedded language.

One of the questions which arises is whether conventional compilation techniques, such as global analysis and resulting transformations, can be applied in the context of EDSLs. The approach we take is that, instead of mapping the embedded language directly onto a function, we first build a representation of the abstract syntax tree of the embedded program fragment. This syntax tree is subsequently analyzed and transformed, and finally mapped onto a function representing its denotational semantics. In this way we achieve run-time "compilation" of the embedded language.

Run-time transformations on embedded languages can have a huge effect on performance. In previous work [63] we present a case study comparing the *Read* instances generated by Haskell's **deriving** construct with instances on which run-time grammar transformations (precedence resolution, left-factorization and left-corner transformation) have been applied.

In this paper we present the library, which has an arrow like interface, which supports in the construction of analyses and transformations, and we demonstrate its use in implementing common sub-expression elimination. The library uses *typed abstract syntax* to represent fragments of embedded programs containing variables

and binding structures, while preserving the idea that the type system of the host language is used to emulate the type system of the embedded language. The tricky issue is how to keep a collection of mutually recursive structures well-typed while it is being transformed.

We finally discuss the typing rules of Haskell, its extensions and those as implemented by the GHC and show that pure System-F based systems are sufficiently rich to express what we want to express, albeit at the cost of an increased complexity of the code.

5.1 Introduction

Modern functional languages such as Haskell and ML are excellent tools for embedding domain specific languages. This is usually done by defining a library of combinators, each combinator representing a grammatical structure from the embedded language. With such a library a programmer can use a domain-specific notation, and at the same time benefit from all the features, such as the type system and abstraction mechanisms, of the general purpose host language. The tight integration of the domain-specific language with the host language has great benefits: there is no need to extend the compiler or to use ad hoc external tools which map the specific notation onto the host language.

One of the essential aspects of a combinator-based domain-specific embedded language is that it shares its type system with the host language.

Although such a combinator-based approach for implementing domain-specific languages looks very convincing at first sight, many problems show up when one starts to use this technique in practice. Just as conventional compilers may analyze programs extensively and may transform them based on the results of such analyses, one also wants to use such techniques for combinator-based embedded languages.

Unfortunately, such analyses and transformations are usually not possible, because the value being constructed by the combinators is a (possibly higher order) function (as in denotational semantics), and hence the internal structure can be neither inspected nor transformed. The approach we take is to have the combinators build a data structure which corresponds to the *typed abstract syntax tree* as described by the grammar of the embedded language. This representation can then be analyzed, transformed, and finally mapped onto its semantics. The main complication we now face is that we have to do this *in a typed setting*.

The problem becomes even harder when the embedded language not only borrows the type system and the abstractions provided by the host language, but also its *declarative* structure. Now we have to deal with collections of abstract syntax trees containing references to each other and the question arises how to representing and observe the binding structures.

In their work on typed meta-programming [45], Pasilic and Linger, using the encoding of equality types in Haskell [5], show how to represent typed ab-

stract syntax trees containing typed references to values. A group of mutually recursive bindings is represented by a nested cartesian product of terms. The variables occurring in the terms are represented by typed pointers into this environment. A similar approach was developed slightly earlier by Xi and Chen, using a dependently typed version of ML, in their work on typeful program transformations [11]. The key ingredient for both is the encoding of environments and references pointing into these environments in host language terms. This encoding ensures that references always refer to *existing values* with the right types. In all approaches, references are basically indices into an environment, encoded as Peano numbers.

In earlier work we demonstrated the practical use of run-time typed grammar transformations for combinator parsers. In [7] we have described a quite involved grammar transformation, the *Left_Corner transform* which effectively removes left-recursive non-terminals from a grammar and replaces them by a set of non left-recursive ones. As a consequence top-down parsers can directly be generated from the resulting grammar. In [63] we eliminate common prefixes from sets of productions in order to improve parsing efficiency. In a small case study we derive an efficient version for the Haskell function *read*, converting the worst case exponential time algorithm used thus far into a linear one.

In this paper we explain the inner-workings of the library underlying these transformations. It has an *Arrow*-style interface and handles introductions of new typed bindings. It is not geared towards grammar transformations and thus can be used to implement a wide variety of other typed program transformations. As a small example of the use of the library we include an implementation of common-subexpression elimination, assuming that the algorithm used thus does not become a object of study by itself and attention can focus on the way the library is used.

Internally, the library makes heavy use of universally and existentially quantified types in combination with Generalized Algebraic Data Types (GADT). This paper therefore can also be seen as a non-trivial exercise using these type system extensions. Throughout the paper, we use GHC [21] syntax for these extensions to Haskell. The code in this paper can be found at: `http://www.cs.uu.nl/wiki/Center/TTTAS`, and was produced by running `lhs2TeX` on the source of this paper. The code is accepted by the Glasgow Haskell Compiler (GHC).

This paper is organized as follows. In Section 5.2, summarizing earlier work [45, 5, 6], we discuss the encoding of typed abstract syntax trees, references and environments. These are the objects for our transformations. In Section 5.3, we develop the library that maintains a changing typed environment. It ensures that all typed references (references that "know" the type of the values they refer to) remain consistent whenever a new definition is added to the environment. In Section 5.4, we show an example of the use of the library: the implementation of common sub-expression elimination. The implementation of the library makes use of lazy-pattern matching on data constructors involving existential types. This combination is unfortunately not supported by GHC. In Section 5.5 we present several solutions to avoid this problem. Finally, in Section 5.6, we

present our conclusions.

5.2 Typed References and Environments

We start by shortly repeating the ideas behind typed meta-programming, in which we represent programs explicitly, so they can be manipulated. We want to do this in such a way that we keep the nice properties that typed programming languages have. Specifically, the fact that the representation is type correct can be seen as a proof that the represented expression is type correct.

As an example consider the abstract syntax (formulated as a GADT) of a simple expression language:

```
data Exp a where
  IntVal   :: Int                                  → Exp Int
  BoolVal  :: Bool                                 → Exp Bool
  Add      :: Exp Int  → Exp Int                   → Exp Int
  Cons     :: Exp a    → Exp [a]                   → Exp [a]
  Nil      ::                                        Exp [a]
  LessThan :: Exp Int  → Exp Int                   → Exp Bool
  If       :: Exp Bool → Exp a → Exp a             → Exp a
```

where the expression

if $3 + 1 < 4$ **then** 5 **else** $1 + 2$

is represented by:

```
expr :: Exp Int
expr = If (LessThan (Add (IntVal 3) (IntVal 1))
                    (IntVal 4))
          (IntVal 5)
          (Add (IntVal 1) (IntVal 2))
```

The value of the represented expression has type *Int*, which is reflected in the type of *expr* :: *Exp Int*. Note that the ill-typed expression $3 <$ *True* cannot be represented, since it will not pass the type-checker.

The type *Exp a* encodes the typing judgement $\vdash e : \alpha$, that reads "the expression e has type α". Each constructor has the structure of a formal judgment. For example *LessThan* encodes the rule:

$$\frac{\vdash e1 : Int \qquad \vdash e2 : Int}{\vdash e1 < e2 : Bool}$$

Before discussing analyses and transformations we have to decide how to represent variables and binding structures. We extend our simple expression language with a constructor *Var*, where a variable is represented by a reference

of type *Ref a env*, an index pointing to a value of type a in an environment of type *env*. In the next section, we will delve into the details of the type *Ref*.

We extend the labelling (properties) of the type *Expr* by an extra type parameter *env*, which stands for the type of the environment in which the expression is to be evaluated:

```
data Expr a env where
  Var     :: Ref a env                          → Expr a env
  IntVal  :: Int                                → Expr Int env
  BoolVal :: Bool                               → Expr Bool env
  Cons    :: Expr a env    → Expr [a] env       → Expr [a] env
  Nil     ::                                      Expr [a] env
  Add     :: Expr Int env  → Expr Int env       → Expr Int env
  LessThan:: Expr Int env  → Expr Int env       → Expr Bool env
  If      :: Expr Bool env → Expr a env
                           → Expr a env         → Expr a env
```

Now we have the judgement $\Gamma \vdash e : \alpha$, that reads "the expression e has type α in local context Γ".

An evaluator *eval* for our simple expression language takes as arguments the abstract syntax tree describing an expression, and an environment which provides values for the variables occurring in the expression, and returns the value of the expression. For the time being only the type of the function *lookup* matters:

```
lookup :: Ref a env → env → a


eval :: Expr a env      → env → a
eval (Var r)          e = lookup r e
eval (IntVal i)       _ = i
eval (BoolVal b)      _ = b
eval (Add x y)        e = eval x e + eval y e
eval (Cons x y)       e = eval x e : eval y e
eval Nil              _ = []
eval (LessThan x y)   e = eval x e < eval y e
eval (If x y z)       e = if eval x e then eval y e else eval z e
```

5.2.1 Type equality

Pasalic and Linger [45] introduce an encoding of typed references that can be used for meta-programming. This encoding relies on the equality type [5, 67, 12]. A (non-diverging) value of type *Equal a b* is a witness of the proof that the types a and b are equal. This witness takes the form of a conversion function, which turns out to always be the identity function.

The addition of GADTs [47] to GHC makes programming with the *Equal* data type a lot easier, because all the fiddling with proofs is implicitly done by

the compiler. Furthermore, the performance increases, since the construction of the proofs is no longer done at run-time. The compiler "knows" that all proofs of type equality are witnessed by values like *id*, *id id*, *id* (*id id*), *id id id* etc, and can thus omit them safely from the generated code: they have no other observable effect than taking time to execute.

The encoding of type equality becomes trivial when using GADTs:

```
data Equal :: * → * → * where
  Eq :: Equal a a
```

The type *Equal* has just one constructor $Eq :: Equal\ a\ a$. If a pattern match on a value of type $Equal\ a\ b$ succeeds (i.e., a non-$\bot$ value *Eq* is available), then the type checker is thus informed that the types a and b were known to be the same at the place the *Eq* was produced.

5.2.2 Typed References

In their paper on typed meta-programming, Pasalic and Linger introduced the data type *Ref* for representing *typed* indices which are labelled with both the type of value to which they refer and the type of environment (a nested cartesian product, growing to the right) in which this value lives.

```
data Ref a env where
  Zero ::                  Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)
```

In the case of a *Suc* we are not interested in the first element; this constructor is polymorphic in the type b. The rules encoded by the type *Ref* are:

$$\frac{}{\Gamma', \alpha \vdash Zero : \alpha} \qquad \frac{\Gamma' \vdash r : \alpha}{\Gamma', \beta \vdash Suc\ r : \alpha}$$

Two references can be compared for equality using the function ($\sim$). If they refer to the same element in the environment this function returns the value *Just Eq*, thus expressing the fact that the types of the referred values are the same too:

```
(∼) :: Ref a env → Ref b env → Maybe (Equal a b)
(∼) Zero    Zero    = Just Eq
(∼) (Suc x) (Suc y) = (∼) x y
(∼) _       _       = Nothing
```

The *lookup* function, the type of which we have seen before, uses its *Ref* parameter as an index in the environment parameter. Whenever we decrease the index, we take the *fst* part of the tuple, until the index reaches *Zero*. The types guarantee that the lookup succeeds:

$$
\begin{aligned}
&\mathit{lookup} :: \mathit{Ref}\ a\ \mathit{env} \to \mathit{env} \to a\\
&\mathit{lookup}\ \mathit{Zero}\ \ \ (\_, a) = a\\
&\mathit{lookup}\ (\mathit{Suc}\ r)\ (e, \_) = \mathit{lookup}\ r\ e
\end{aligned}
$$

The function *update* takes an additional function as argument, which is used to update the value the reference addresses. The other values in the environment are left unchanged:

$$
\begin{aligned}
&\mathit{update} :: (a \to a) \to \mathit{Ref}\ a\ \mathit{env} \to \mathit{env} \to \mathit{env}\\
&\mathit{update}\ f\ \mathit{Zero}\ \ \ (e, a) = (e, f\ a)\\
&\mathit{update}\ f\ (\mathit{Suc}\ r)\ (e, x) = (\mathit{update}\ f\ r\ e, x)
\end{aligned}
$$

As an example, consider the *example* environment:

$$
\begin{aligned}
&\textbf{type}\ \mathit{ExampleEnv} = ((((), \mathit{Int}), \mathit{Char}), \mathit{String})\\
&\mathit{example} = ((((), 1), \texttt{'a'}), \texttt{"b"}) :: \mathit{ExampleEnv}\\
&\mathit{ref}_a \quad = \quad \mathit{Suc}\ \mathit{Zero} \quad :: \mathit{Ref}\ \mathit{Char}\ \mathit{ExampleEnv}\\
&\mathit{ref}_one \quad = \mathit{Suc}\ (\mathit{Suc}\ \mathit{Zero}) \quad :: \mathit{Ref}\ \mathit{Int}\ \ \mathit{ExampleEnv}
\end{aligned}
$$

The expression *lookup* ref_a *example* yields the character `'a'`, while the expression *lookup* ref_one *example* yields the integer 1. Notice that the type of the reference determines the type of the result! Application of *update* $(+5)$ ref_one to the example environment updates it to $((((), 6), \texttt{'a'}), \texttt{"b"})$. This clearly shows that the ref_a and ref_one refer to values of different types in the same environment.

With our extended data type we now also can encode expressions which contain variables of different types, such as **if** b **then** 3 **else** a, using an environment containing an *Int* and a *Bool*:

$$
\begin{aligned}
&\mathit{var\_a} = \mathit{Var}\ (\mathit{Suc}\ \mathit{Zero})\\
&\mathit{var\_b} = \mathit{Var}\ \ \ \ \mathit{Zero}\\
&e \quad = \mathit{If}\ \mathit{var\_b}\ (\mathit{IntVal}\ 3)\ \mathit{var\_a} :: \mathit{Expr}\ \mathit{Int}\ (((), \mathit{Int}), \mathit{Bool})\\
&\mathit{env} \quad = (((), 11), \mathit{False}) \quad :: \quad (((), \mathit{Int}), \mathit{Bool})\\
&\mathit{test} \quad = \mathit{eval}\ e\ \mathit{env} \quad :: \quad \mathit{Int}
\end{aligned}
$$

Some may complain that this Peano representation is extremely cumbersome and error prone. In [6], we have shown how, by using some extra combinators, this problem can be overcome. Furthermore the type system also helps us to avoid accidental mistakes. Also, note that building and maintaining the internal representation is the work of the combinator library and is largely invisible to the programmer describing program transformations.

5.2.3 Declarations

In this section we will focus on the problem how to represent a collection of possibly mutually recursive definitions, each consisting of an identifier being defined and a right-hand side expression containing these identifiers.

The idea is to store the right-hand side expressions in a heterogeneous list, and represent the identifiers by indices in this list. This is very similar to the environments described above, with the main difference that the actual environment now contains abstract syntax terms labelled with a type instead of values having a type.

In this setting an environment consists of terms which in turn are labelled with the type of the environment, because this type is used to label the references contained in these terms. Representing such environments as a nested tuples would lead to an infinite type. For example consider the following two declarations:

$$\begin{aligned}&\textbf{let } x = 1 : y\\ &\quad\ \ y = 2 : x\end{aligned}$$

These declarations give rise to an environment containing two terms. Suppose we name the type of this environment *TwoLists*, then both terms in the environment have type $Expr\ [Int]\ TwoLists$. This leads to the following type for the environment:

$$\begin{aligned}\textbf{type } TwoLists = (((), &Expr\ [Int]\ TwoLists)\\ &, Expr\ [Int]\ TwoLists)\end{aligned}$$

This type definition is cyclic, and is thus not allowed in Haskell.

Our solution is found in splitting the *env* type parameter into two parameters: one for the environment addressed by the references occurring in the terms and one describing the environment which is being constructed by the sequence of terms. The type $Env\ term\ use\ def$ represents a sequence of instantiations of type $\forall\, a\ .\ term\ a\ use$, where all the instances of a are stored in the type parameter *def*; thus the type *def* contains the type parameters a of the terms of type $term\ a\ use$ occurring in the $Env\ term\ use\ def$. The type *use* on the other hand is a sequence containing the types to which may be referred from within terms of type $term\ a\ use$.

$$\begin{aligned}&\textbf{data } Env\ term\ use\ def\ \textbf{where}\\ &\quad Empty\ ::\ Env\ t\ use\ ()\\ &\quad Ext\quad ::\ Env\ t\ use\ def' \rightarrow t\ a\ use\\ &\qquad\ \ \rightarrow Env\ t\ use\ (def', a)\end{aligned}$$

When the types *def* and *use* coincide we can be sure that the references in the terms do not point to values outside the environment and do point to terms representing the right type. hence we can use the environment being defined as the environment to be indexed by the references contained in the right-hand side terms of the definitions.

Splitting this single type into two type parameters, which we only require in the end to be equal, makes it possible to to use references which refer to terms which still have to be added. Only after we are done with manipulating and extending the environment we require *use* and *def* to be the same! The fact

that a sequence of terms is closed and well-typed is thus encoded in the type system of the host language. So the mutually recursive declarations:

$$\begin{aligned}&\textbf{let } x = 1 : y\\&\quad\ \ y = 2 : x\end{aligned}$$

is encoded as:

$$\begin{aligned}&\textbf{type } \mathit{Final} = (((), [\mathit{Int}]), [\mathit{Int}])\\&x \quad = \mathit{Var}\ (\mathit{Suc}\ \mathit{Zero}) \quad :: \quad \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final}\\&y \quad = \mathit{Var}\ \mathit{Zero} \quad :: \quad \mathit{Expr}\ [\mathit{Int}]\ \mathit{Final}\\&\mathit{decls} :: \mathit{Env}\ \mathit{Expr}\ \mathit{Final}\ \mathit{Final}\\&\mathit{decls} = \mathit{Empty}\ \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{IntVal}\ 1)\ y\\&\qquad\qquad\ \ \text{`}\mathit{Ext}\text{`}\ \mathit{Cons}\ (\mathit{IntVal}\ 2)\ x\end{aligned}$$

where we note that the x and y here are Haskell values referring to the right-hand side terms of their definitions in the *Env*.

The lookup and update operations on the data type *Env* are defined in a similar way as before:

$$\begin{aligned}&\mathit{lookupEnv} :: \mathit{Ref}\ a\ \mathit{env} \to \mathit{Env}\ t\ s\ \mathit{env} \to t\ a\ s\\&\mathit{lookupEnv}\ \mathit{Zero}\ \ (\mathit{Ext}\ \_\ t) = t\\&\mathit{lookupEnv}\ (\mathit{Suc}\ r)\ (\mathit{Ext}\ \mathit{ts}\ \_) = \mathit{lookupEnv}\ r\ \mathit{ts}\\&\mathit{updateEnv} :: (t\ a\ s \to t\ a\ s) \to \mathit{Ref}\ a\ \mathit{env} \to \mathit{Env}\ t\ s\ \mathit{env} \to \mathit{Env}\ t\ s\ \mathit{env}\\&\mathit{updateEnv}\ f\ \mathit{Zero}\ \ (\mathit{Ext}\ \mathit{ts}\ t) = \mathit{Ext}\ \mathit{ts}\ (f\ t)\\&\mathit{updateEnv}\ f\ (\mathit{Suc}\ r)\ (\mathit{Ext}\ \mathit{ts}\ t) = \mathit{Ext}\ (\mathit{updateEnv}\ f\ r\ \mathit{ts})\ t\end{aligned}$$

The chosen representation now has an efficiency problem, to be fixed in the next section: whenever we extend the environment with a new *Ext* all existing references occurring in terms already stored in the environment have to be incremented by applying an extra *Suc* constructor to them, since the values to which they refer have an index that is one higher in the new environment.

5.3 Transformation Library

In this section we develop a type *Trafo* representing typed transformation steps on a heterogeneous collection and an *Arrow*-like [30] library of combinators for composing such transformations. Each *Trafo* takes input and produces output and can be composed in the same way as *Arrow*s. Internally it maintains an environment containing abstract syntax terms. Additionally, meta-data about the transformation process can be maintained as well. Such meta-data could for example be a symbol table, debugging information, reference counts, etc.

In developing the type *Trafo* we use a Haskell-like type synonym syntax augmented with the symbols $\forall$ and $\exists$ to denote universally and existentially quantified types. We first develop the type *Trafo* to tackle the problem of maintaining a heterogeneous collection of definitions, and subsequently extend

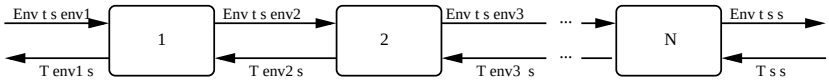

Figure 5.1: Dataflow between transformation steps

it with *Arrow*-style inputs, outputs, and meta-data. Finally we encode the *Trafo* using data types, as accepted by GHC.

We model a collection of embedded-language definitions as a value of type *Env*, and make these definitions the subject of transformations that may induce new definitions, as in the case of common-subexpression removal where subexpressions get named. At the end of the transformation process each reference occurring somewhere in a term stored in this *Env* must be a reference into the final set of definitions. In this case, we call the environment *closed*. One way to ensure that our environment is always closed is to adjust all the references in all the terms whenever a new definition is added to the environment. Unfortunately new definitions are to be added at the head of the sequence (i.e. at position *Zero*), which implies that all existing references have to be updated by the application of an extra *Suc* constructor. This is cumbersome and inefficient, and is better done once, i.e., when we know how many *Suc*s to add to each reference so it addresses the right element in the final structure.

We only require an environment to be closed after all transformations have been applied and all new definitions have been added. The final type must be of the form *Env t s s* (or *FinalEnv t s*) for some type *s*.

$$\textbf{type}\ \mathit{FinalEnv}\ t\ \mathit{usedef} = \mathit{Env}\ t\ \mathit{usedef}\ \mathit{usedef}$$

References into this environment *s* are coined *final references*. If all the transformation steps only add terms of type $(\exists a\ .\ t\ a\ s)$ to the environment, then they contain only final references, and we do not need to adjust the references after each transformation step. However, this seems to be impossible. How can we make the transformation steps half way through the transformation process construct terms of the type $(\exists a\ .\ t\ a\ s)$? After all *s* is the type of the final environment and is only known after all transformation steps have completed. For creating such final references we need to know how many new definitions will be added to the environment by future transformation steps.

We solve this problem by using Haskell's lazy evaluation to pass knowledge about the "future" backwards through the computation. In our case information about the number of definitions added by future steps is encoded in a *Ref*-transformer, that prepends as many *Suc*-nodes to a reference as there are new definitions to come. The type of such *Ref*-transformers is[1]:

$$\textbf{data}\ T\ e\ s = T\ \{\mathit{unT} :: \forall x\ .\ \mathit{Ref}\ x\ e \to \mathit{Ref}\ x\ s\}$$

[1]Note that the keyword **forall** is presented by the logical symbol ∀

Figure 5.1 depicts the idea described above. The environment is constructed from left to right. Each step takes as input the environment constructed thus far and yields an updated environment as result. On the top left, the computation starts with an environment of some type *Env t s env1*. Each step extends the environment with some new definitions, making the type of the environment change at every step. The final result of all steps has to be an environment of type *FinalEnv t s*. *Ref*-transformers are passed on and modified from right to left. These transformer effectively inform every intermediate step how deep down the environment constructed thus far is located in the final environment.

The environment yielded by the last step is the place where the *use* and the *def* types have to coincide. Therefore the identity transformer is used as the initial value for the "pass-back" chain. Every step updates the transformer according to the number of definitions it adds to the environment, before passing it on to its preceding transformation step.

5.3.1 The *Trafo* data type

We now develop the type *Trafo* to implement the idea described above. Every step has two incoming and two outgoing arrows, one of each type at each side. This means our *Trafo*-type is a function taking two arguments and returning two results. We want our *Trafo*-type to be polymorphic in the type of the terms (t) stored in the environment. As a first attempt, we take:

$$\textbf{type}\ \mathit{Trafo}\ t = T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \to (T\ \mathit{env1}\ s, \mathit{Env}\ t\ s\ \mathit{env2})$$

the role of the different elements is as follows:

Env t s env1 is the environment which has been constructed up to where the current transformation starts, and corresponds to the incoming arrow at the top left. The *env1* parameter describes which elements have thus far been added to the environment.

T env2 s is the incoming arrow at the bottom right. It maps references into an environment labelled with *env2* into references into the final environment *s*.

Env t s env2 is the environment constructed by this step, and *env2* will usually be either *env1* or an extension of *env1*.

T env1 s corresponding to the bottom left arrow coming out of a *Trafo*, is the updated *T env2 s*, which can be constructed by the transformation since it knows how many elements it adds to the environment.

This type definition is incomplete, the type variables *env1*, *env2*, and *s* are still unbound. We do not want all these variables to appear on the left-hand side of the type definition, as this would expose the internal complexity of the library to the user, so we have to add universal or existential quantifiers.

The type *env1* is the type of the environment constructed thus-far. A step should not make any assumptions about this environment, and hence the type

variable *env1* is to be universally quantified. The type *env2* is the type of the result of a transformation step. This type depends on the number of new definitions introduced by the step. As this can be an arbitrary number, the type *env2* is fully determined by the transformation and the incoming *env1*. Because *env2* depends on *env1* by extending it, this quantifier has to be within the scope of *env1*: once *env1* is fixed the transformation fixes *env2*. Finally, the type variable s represents the type of the final result, which we do expose. Its role is similar to the s in the type $ST\ s\ a$, which is the type of state threads [36] of the Haskell libraries. All this leads to the following definition for the type *Trafo*:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Trafo}\ t\ s = \\
&\quad \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\quad T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
&\qquad\qquad\qquad \to (T\ \mathit{env1}\ s,\quad \mathit{Env}\ t\ s\ \mathit{env2})
\end{aligned}
$$

In the next step, we extend the type *Trafo* with *Arrow*-style input and output. This allows us to pass values from one transformation *Trafo* to the next, in addition to the implicitly passed environment and ref-transformers. So we add two arguments (a and b) to the type *Trafo*, which stand for the types of the input and output:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Trafo}\ t\ s\ a\ b = \\
&\quad \forall \mathit{env1}\ .\ \exists \mathit{env2}\ .\quad a \to T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
&\qquad\qquad\qquad \to (b,\quad T\ \mathit{env1}\ s,\quad \mathit{Env}\ t\ s\ \mathit{env2})
\end{aligned}
$$

Finally we may want to maintain meta-information about the environment, such as which elements have been added already. This information may be used to determine whether new elements have to be added to the environment; hence it has to live outside the type which is existentially quantified by *env2*, but it will in general depend on the environment *env1* constructed thus far.

Thus, we introduce an extra argument m that stands for the type of the meta-data. A *Trafo* takes the meta-data on the current environment *env1* as input and yields meta-data for the (possibly extended) environment *env2*.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Trafo}\ m\ t\ s\ a\ b = \\
&\quad \forall \mathit{env1}\ .\ m\ \mathit{env1} \\
&\qquad \to \exists \mathit{env2}\ . \\
&\qquad\quad (\ m\ \mathit{env2} \\
&\qquad\quad ,\quad a \to T\ \mathit{env2}\ s \to \mathit{Env}\ t\ s\ \mathit{env1} \\
&\qquad\qquad \to (b,\quad T\ \mathit{env1}\ s,\quad \mathit{Env}\ t\ s\ \mathit{env2}) \\
&\qquad\quad)
\end{aligned}
$$

We now have come to a problematic point: the type above is not Haskell, nor is it accepted by the GHC due to the use of existential quantifiers in type definitions. Unfortunately an existential type can only be introduced by using the keyword $\forall$ on the left side of a constructor in a data-declaration. Thus, we have to resort to an encoding of the above type using two other data types:

Figure 5.2:

```
data Trafo m t s a b =
  Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
  ∃env2 . TrafoE (m env2)
                 (a → T env2 s → Env t s env1
                  → (b, T env1 s, Env t s env2)
                 )
```

Now that we have developed the final version of our *Trafo* data type we can define the combinators to construct and compose transformation steps.

5.3.2 Creating new references

The most important operation is the extension of the environment with a new term, returning a reference to this newly added term. This operation is implemented by *newSRef*, which takes a typed term as input, adds it to the environment and yields a reference pointing to this value. The type of *newSRef* is:

```
newSRef :: Trafo Unit t s (t a s) (Ref a s)
data Unit s = Unit
```

No meta-information on the environment is recorded by *newSRef*; therefore we use the type *Unit* for the meta-data. If meta-information is required, one must define an application-specific version of *newSRef*, as we will do in our example application in Section 5.4. The type variable t stands for the type of the terms. We want the input to *newSRef* to be of type $t\ a\ s$, where s stands for the type of the final environment. The result of a *newSRef* is a reference of type $Ref\ a\ s$, which points to the newly inserted, a labelled, term in the final environment.

```
newSRef
  = Trafo (λ_ → TrafoE Unit extEnv)
extEnv   ::   t a s →   T (e, a) s → Env t s e
          → (Ref a s,   T e      s,    Env t s (e, a))
extEnv
  = λt_a (T tr) env → (tr Zero, T (tr . Suc), Ext env t_a)
```

The incoming meta-information is ignored and *Unit* is returned as meta-data. The function *extEnv*, used in *TrafoE*, is more interesting: it takes

as arguments the term to be inserted $t_a :: t\ a\ s$, the current environment $env :: Env\ t\ s\ e$, and a reference transformer $(T\ tr) :: T\ e\ s$, which transforms references into the current environment into references into the final environment. The result is a tuple containing the new environment, which has type $Env\ t\ s\ (a, e)$, and a reference of type $Ref\ a\ s$. The term (t_a) becomes the last element of the new environment, hence the reference pointing to this term is *Zero*. However, more terms may be added in the future, therefore the reference transformer (tr) is applied, which basically prepends to *Zero* as many *Suc*-nodes as there are future additions to the environment under construction. Finally, we record the fact that one new element was added to the environment by adding an extra *Suc*-node to the reference transformer tr, which we pass on to our predecessors. Since all application-specific versions of *newSRef* have to do this work, we include the function *extEnv* as part of the library.

In certain cases an application-specific *newSRef* will not have to add a new term, but will return an existing reference instead. For these cases we added a function that casts a reference in the constructed environment to one in the final environment:

$$\begin{aligned}
&castSRef\ ::\ \ Ref\ a\ env \\
&\qquad\qquad \rightarrow\ \ x\ \ \ \ \ \ \ \ \rightarrow T\ env\ s \rightarrow Env\ t\ s\ env \\
&\qquad\qquad \rightarrow\ (Ref\ a\ s\ \ ,\ \ T\ env\ s\ \ ,\ \ Env\ t\ s\ env) \\
&castSRef\ r = (\lambda_\ (T\ t)\ decls \rightarrow (t\ r, T\ t, decls))
\end{aligned}$$

5.3.3 *runTrafo*

Of course we want to "run" our *Trafo*-computations. This is the task of the function *runTrafo*, which has the following type:

$$runTrafo :: (\forall s\ .\ Trafo\ m\ t\ s\ a\ (b\ s)) \rightarrow m\ () \rightarrow a \rightarrow Result\ m\ t\ b$$

$$\textbf{data}\ Result\ m\ t\ b = \exists s\ .\ Result\ (m\ s)\ (b\ s)\ (FinalEnv\ t\ s)$$

The type of *runTrafo* is inspired by that of $runST :: (\forall s\ .\ ST\ s\ a) \rightarrow a$, which is part of the state thread library (ST). The rank-2 type for *runTrafo* ensures that transformation steps cannot make any assumptions about the type of final environment (s).

The function *runTrafo* takes as arguments the *Trafo* we want to run, meta-information for the empty environment, and an input value. The result of *runTrafo* is the final environment $(Env\ t\ s\ s)$ together with the resulting meta-data $(m\ s)$, and the output value $(b\ s)$. Because s could be anything we have to hide it using existential quantification, and thus introduce the data definition *Result*.

Note that the type of the output is $(b\ s)$, one might wonder why the output is not just some type c(not labelled with s). The reason is that returning a value of type $b\ s$ is slightly more general. It allows a transformation to return a value labelled with type s, which would otherwise not be allowed.

The implementation of the function *runTrafo* reads:

```
runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a → Result m t b
runTrafo trafo m a =
  let Trafo trf      = trafo
      TrafoE m2 f = trf m
  in  case f a (T id) Empty of (b, _, env) → Result m2 b env
```

The function f inside the *Trafo* type is applied to the input value (a), the identity *Ref*-transformer, and the empty environment. The result of f is the output (b), and the transformation result (env), which are wrapped in a *Result* constructor together with the resulting meta-data ($m2$). The function *runTrafo* uses lazy pattern binding for the matches on *Trafo* and *TrafoE* (for strict pattern matching one should use **case** instead of **let**). This is essential as we need to instantiate the universally quantified s with the existential type constructed by the inner *TrafoE* constructor. Unfortunately this is not allowed by the Glasgow Haskell compiler (GHC) as it forbids the use of lazy pattern matching in combination with existential types. Other compilers such as Hugs[32], and the Haskell compiler under construction at Utrecht University[62] do allow this combination. In Section 5.5 we suggest two solutions to circumvent this problem.

5.3.4 Arrow-style combinators

The *Arrow* library consists of a set of functions for constructing and combining values that are instance of the *Arrow* class. Furthermore there is a convenient notation for programming with *Arrow*s. This notation is inspired by the **do**-notation for *Monad*s. To implement the *Arrow* interface one needs to implement three methods *arr*, >>>, and *first*.

We make the type (*Trafo m t s*) instance of the *Arrow* class:

```
instance Arrow (Trafo m t s) where
```

The method *arr* lifts a function.

```
  -- arr :: (a → b) → Trafo m t s a b
arr f = Trafo (λm → TrafoE m (λa t e → (f a, t, e)))
```

The >>> operator composes two *Trafo*s[2]. It is actually a straightforward transcription of the composition depicted in Figure 5.1. In that figure box 1 refers to the incoming environment, box 2 to the intermediate and box 3 to the outgoing.

```
  -- (>>>) :: Trafo m t s a b → Trafo m t s b c → Trafo m t s a c
Trafo t1 >>> Trafo t2 = Trafo
  (λm1 → case t1 m1 of
    TrafoE m2 f1 → case t2 m2 of
```

[2]Since GHC 6.10 the operator >>> is no longer part of the *Arrow* class. A modified version of the code in this chapter can be found at `http://hackage.haskell.org/package/TTTAS`.

```
        TrafoE m3 f2 →
          TrafoE m3 (λa tt env1 → let (b, tt1, env2) = f1 a tt2 env1
                                      (c, tt2, env3) = f2 b tt env2
                                  in  (c, tt1, env3)
                    )
  )
```

The method *first* applies the first component of the input to the argument *Trafo* and copies the rest unchanged to the output. It is implemented as follows:

```
  -- first :: Trafo m t s a b → Trafo m t s (a, c) (b, c)
first (Trafo tr)
  = Trafo (λm1 → case tr m1 of
    TrafoE m2 f →
      TrafoE
      m2
      (λ∼(a, c) tt env1 →
        let (b, tt1, env2) = f a tt env1
        in ((b, c), tt1, env2)))
```

For easy reference, we also show the other functions of the *Arrow*-interface. The code is just the default definition found in the *Arrow*-class.

```
second  :: Trafo m t s b c → Trafo m t s (d, b) (d, c)
second f = arr swap >>> first f >>> arr swap
  where swap∼(x, y) = (y, x)

(***)  :: Trafo m t s a b → Trafo m t s c d → Trafo m t s (a, c) (b, d)
f *** g = first f >>> second g
(&&&)  :: Trafo m t s a b → Trafo m t s a c → Trafo m t s a (b, c)
f &&& g = arr (λa → (a, a)) >>> (f *** g)
```

The function *loop* is used to construct feedback loops. It takes a *Trafo* that has an input of type (a, x) and output of type (b, x). The component of type x is fed back resulting in a *Trafo* with input a and output b.

```
instance ArrowLoop (Trafo m t s) where
    -- loop :: Trafo m t s (a, x) (b, x) → Trafo m t s a b
  loop (Trafo st) =
    Trafo
    (λm → case st m of
      TrafoE m1 f1 →
        TrafoE m1
        (λa t e →
          let ((b, x), t1, e1) = f1 ((a, x)) t e
          in (b, t1, e1)
        )
    )
```

5.4 Common sub-expression elimination

In this section we show how the library developed in the previous section can be applied to implement common sub-expression elimination (CSE). The object of this transformation is the *Expr*-language from Section 5.2.

CSE is a compiler optimization, where for each sub-expression e that occurs more than once the CSE transformation introduces a new declaration $v = e$, and furthermore replaces all subsequent occurrences of e with the variable v.

For example the following expressions:

$$a = 4;$$
$$b = (a + 4) + (a + 4);$$

are transformed into:

$$a = 4;$$
$$x = a + a;$$
$$b = x + x;$$

The subject of our *CSE* transformation is a sequence of (possibly mutually recursive) declarations. These are represented as an *Env* of typed *Expr*s:

$$\textbf{type}\ \mathit{Decls}\ \mathit{env} = \mathit{Env}\ \mathit{Expr}\ \mathit{env}\ \mathit{env}$$

In the type *Decls* above the type variable *env* encodes the type of each of the declarations. The result of the transformation is also a sequence of declarations. It is labelled with a different type variable because the *CSE* transformation may introduce new declarations. The amount of newly introduced declarations depends on the number of common sub-expressions in the original set of declarations. As a result the type of the result of the transformation is not statically known. Therefore we introduce the following existential type for the result of the *CSE* transformation:

$$\begin{aligned}\textbf{data}\ \mathit{TDecls}\ \mathit{env} = \exists \mathit{env}'\ .\ \mathit{TDecls}\ &(\mathit{Decls}\ \mathit{env}') \\ &(T\ \mathit{env}\ \mathit{env}')\end{aligned}$$

In the type *TDecls env*, the type variable *env* stands for the type of the original declarations. The type *TDecls* contains a sequence of declarations (*Decls env*′), in which the type variable *env*′ represents the type of the transformed declarations. The transformed declarations are accompanied by a *Ref*-transformer mapping references from the original sequence of declarations to references in the new one.

Summarizing the type implementation of the *CSE* transform, developed in the remainder of this section has the following type:

$$\mathit{cse} :: \mathit{Decls}\ \mathit{env} \to \mathit{TDecls}\ \mathit{env}$$

Before we delve into the implementation of *cse*, we first show an example.

```
a = 4;
b = (a + 4) + (a + 4);
```

These declarations are encoded as typed abstract syntax as follows:

```
a = Suc Zero
b = Zero
exampledecls :: Decls ((((), Int), Int), Int)
exampledecls =
  Empty `Ext` (IntVal 4)
        `Ext` (Add (Add (Var a) (IntVal 4))
                   (Add (Var a) (IntVal 4)))
```

To transform the declarations we apply the *cse* function:

```
resdecls :: TDecls ((((), Int), Int), Int)
resdecls = cse exampledecls
```

The transformed declarations (*resdecls*) can be used as follows:

```
evalDecls :: Decls env → env
evalVar   :: Ref a env → TDecls env → a
evalVar var (TDecls ds (T tt))
  = lookup (tt var) (evalDecls ds)


value_a = evalVar a resdecls
value_b = evalVar b resdecls
```

The function *evalVar* takes a reference and the transformed declarations as arguments. It evaluates the declarations(*evalDecls*) and uses the reference in combination with the *Ref*-transformer(*tt*) to select the value from the evaluated declaration that corresponds the reference(*var*). Note that we omitted the definition of *evalDecls* and only show its type.

The transformed declarations (*resdecls*) internally have the following structure:

```
TDecls
  (   Empty `Ext` (IntVal 4)
            `Ext` (Add (Var (Suc (Suc Zero)))
                       (Var (Suc (Suc Zero))))
            `Ext` (Add (Var (Suc Zero))
                       (Var (Suc Zero)))
  )
  (   T (λref → case ref of
          Zero      → Zero
          Suc Zero → Suc (Suc Zero))
        :: T ((((), Int), Int), Int) ((((), Int), Int), Int), Int))
```

A new declaration has been inserted in between those for a and b, this fact is reflected in the *Ref*-transformer. The reference *Zero* (for b) remains unchanged because the declaration b is still the last one. The reference for declaration a however gets an extra *Suc* node.

5.4.1 Implementation

Briefly our implementation of *CSE* performs the following steps:
For each sub-expression

- check if we already encountered it
  - if not, add a declaration for this sub-expression to the result
  - if yes, replace it by a reference to the equivalent expression that is already in the result

To determine whether expressions have common sub-expressions we need to compare expressions for equality. Therefore we introduce the function *equals*, which compares two expressions, and, if they are equal returns a witness encoding that the types of the two expressions are the same.

```
equals :: Expr a env → Expr b env → Maybe (Equal a b)
equals (Var r1)    (Var r2)                = r1 ∼ r2
equals (IntVal i1) (IntVal i2)
                     | i1 ≡ i2             = Just Eq
equals (LessThan x1 y1) (LessThan x2 y2) = do Eq ← equals x1 x2
                                              Eq ← equals y1 y2
                                              return Eq
...
equals _           _                     = Nothing
```

The implementation of the function *equals* is fairly straightforward. To determine whether two *Var*s are equal the function $(\sim)$ is applied to determine whether the contained references are the same. Two *IntVal* expressions are equal if their contained values are the same. To determine whether two *LessThan* expressions are equivalent, the function *equals* is recursively applied on their components. We omit the definitions of *equals* for the constructors *BoolVal*, *Add* and *If*, because they are very similar to the ones above.

During the *CSE* transformation we need to determine whether we already encountered an expression before. If an expression has not been encountered before, a declaration for it is added to the result. On the other hand if it was encountered before, it is not added to the result, but is instead replaced by a reference to the equivalent expression that is already present in the result.

For this we introduce the type *Memo*:

```
data Memo env env' = Memo (∀x . Expr x env → Maybe (Ref x env'))
```

The *Memo* tells us whether an expression has been encountered before, and if so, returns a witness in the form of a reference to the copy of the expression in the transformation result. Note the use of two distinct type variables: *env* stands for the type of the original sequence of declarations and *env′* for the result of the transformation.

We introduce a "smart" constructor to create an empty *Memo*:

```
emptyMemo :: Memo env ()
emptyMemo = Memo (const Nothing)
```

We proceed by introducing a type synonym for the *CSE* transformation *Arrow*:

```
type TrafoCSE env = Trafo (Memo env) Expr
```

The terms that are to be transformed have type *Expr* and the state (meta-data) maintained is a table of type *Memo*.

During the transformation all sub-expressions are visited. For each sub-expression we check whether it has already been encountered before. If so the table of type *Memo* provides us a reference to the earlier occurrence of the sub-expression, which is used as a replacement for the current sub-expression. On the other hand if the sub-expression was not encountered before, the *Memo* table is extended with an entry for this sub-expression.

This is captured in the function *insertIfNew*, which is our application specific version of *newSRef*. Its argument is the sub-expression that is being visited. Its result is a *TrafoCSE* with as input the transformed version of the sub-expression, which has type (*Expr a s*). The output is a reference to the transformed version of the first occurrence of the sub-expression.

```
insertIfNew :: ∃s a env . Expr a env → TrafoCSE env s (Expr a s)
                                                      (Ref a s)
insertIfNew e =
  Trafo
  (λ(Memo m :: Memo env env′) → case m e of
     Nothing → TrafoE   (extMemo e (Memo m)) extEnv
     Just r  → TrafoE   (Memo m)             (castSRef r)
  )
extMemo :: Expr a env → Memo env env′ → Memo env (env′, a)
extMemo e (Memo m)
        = Memo (λs → case equals e s of
                       Just Eq → Just Zero
                       Nothing → fmap Suc (m s)
               )
```

The first time we encounter a sub-expression it is not found in the *Memo*-table (i.e. the *Nothing*-case above). Firstly the transformed version of the sub-expression is appended to the transformed declarations using *extEnv*. The *Memo* table is extended (using *extMemo*) with an entry mapping the current

sub-expression to *Zero*, so for a next occurrence of the sub-expression we know where to find the transformed first occurrence. Because we added one declaration ourselves, one extra *Suc* is added to the rest of the references in the *Memo* table.

For every subsequent encounter of the sub-expression, we find it in the *Memo* table (i.e. the *Just* case above). The reference to the first occurrence is simply the one found in the *Memo* table. We apply the function *castSRef* to take into account the declarations that might be added by future transformation steps.

The function *app_cse* [3] applies the *CSE* transformation to a single expression. The resulting *TrafoCSE* has as arrow input a *Ref*-transformer, that maps references from the original sequence of declarations to corresponding ones pointing into the transformation result. As output the *TrafoCSE* yields a reference to the transformed expression.

$$\begin{aligned}&app\_cse :: Expr\ a\ env \to TrafoCSE\ env\ s\ (T\ env\ s)\ (Ref\ a\ s)\\&app\_cse\ (Var\ r) = \mathbf{proc}\ (T\ tenv\_s) \to returnA \prec tenv\_s\ r\end{aligned}$$

The reference inside a variable is transformed by applying the supplied *Ref*-transformer. The transformed reference now points to the corresponding value in the transformation result.

$$app\_cse\ e@(IntVal\ i) = \mathbf{proc}\ \_ \to insertIfNew\ e \prec IntVal\ i$$

For integer constants the function *insertIfNew* is applied to the original expression (e). As transformed expression $IntVal\ \ i$ is passed. The function *insertIfNew* only inserts this expression if the integer constant is not already presented in the transformation result.

$$\begin{aligned}&app\_cse\ e@(LessThan\ x\ y)\\&\quad = \mathbf{proc}\ tt \to \mathbf{do}\ l \leftarrow app\_cse\ x \prec tt\\&\qquad\qquad\qquad\quad r \leftarrow app\_cse\ y \prec tt\\&\qquad\qquad\qquad\quad insertIfNew\ e \prec LessThan\ (Var\ l)\ (Var\ r)\\&\quad \ldots\end{aligned}$$

For the constructor *LessThan* the function *app_cse* is applied recursively resulting in references to the transformed sub-expressions. The *Ref*-transformer *tt* is passed for both sub-expressions. Again *insertIfNew* is applied to the original expression; as transformed expression we pass a *LessThen* node containing the references to the transformed sub-expressions.

The implementations of *app_cse* for the constructors *BoolVal*, *Add*, and *If* are very similar, and are therefore omitted.

The function *app_cse* defined above applies the *CSE* transform to a single expression only. The final transformation should transform a sequence of declarations, which is encoded as a value of the data type *Env*. Therefore we define *cse_env*, which takes an *Env* as argument and applies *app_cse* to each expression. Analogous to *app_cse*, the function *cse_env* takes a *Ref*-transformer as

[3] The following functions use arrow notation [44]

input. It collects all the references returned by *app_cse* in an *Env*. This collection contains for each reference of the original declarations a corresponding reference in the transformation result.

```
cse_env :: Env Expr env env' → TrafoCSE env s (T env s)
                                               (Env Ref s env')
cse_env Empty      = proc _      → returnA ≺ Empty
cse_env (Ext es e) = proc tt     → do renv ← cse_env es ≺ tt
                                      r    ← app_cse e  ≺ tt
                                      returnA ≺ Ext renv r
```

The collection of *Ref*s returned by the function *cse_env* can be used to compute the *Ref*-transformer that it requires as input:

```
refTransformer :: Env Ref s env → T env s
refTransformer refs = T (λr → lookupEnv r refs)
```

The result of *cse_env* is used to compute its own input. To construct such a feedback-loop, we use the special **mdo**-notation for mutually recursive *Arrow* statements.

```
trafo :: Decls env → TrafoCSE env s () (T env s)
trafo decls = proc _ → mdo let tt = refTransformer refs
                           refs ← cse_env decls ≺ tt
                           returnA ≺ tt
```

Finally we present the function *cse* which simply runs the *trafo* and extracts the result:

```
cse :: Decls env → TDecls env
cse decls = case runTrafo (trafo decls) emptyMemo () of
              Result _ t env → TDecls env t
```

5.5 Alternative implementation for *runTrafo*

Recall the data type *Trafo* and the function *runTrafo*:

```
data Trafo m t s a b =
  Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
  ∃env2 . TrafoE
           (m env2)
           (a → T env2 s → Env t s env1 →
              (b, T env1 s, Env t s env2)
           )
```

```
runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a → Result m t b
runTrafo trafo m a =
  let Trafo trf      = trafo
      TrafoE m2 f = trf m
  in  case f a (T id) Empty of
        (b, _, env) → Result m2 b env
```

In the definition of *runTrafo* we want the type of the final environment (s) to be the same as the type of the environment coming out of the transformation (*env2*). To achieve this the universally quantified s must be instantiated as *env2*. For this the use of lazy pattern binding (using **let**) on the existential data type (*TrafoE*) is essential. Unfortunately GHC, the most widely used Haskell Compiler, does not support lazy pattern matching on data constructors with existential types. In such cases it reports the infamous "My brain just exploded" error message. Other compilers such as Hugs[32] and UHC[62] do support lazy pattern matching on data constructors with existential types. The reason this is not supported by GHC, is because it cannot be translated into GHCs intermediate language, which is based on System-F. GHCs core language should be extended with some kind of fix-point operator at the type level. However, this has as disadvantage that type level terms may be non-terminating, and therefore type terms can no longer be simply erased.

Using *unsafeCoerce* is a simple solution for this problem:

```
unsafeCoerce :: a → b
runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a → Result m t b
runTrafo trafo m a = case trafo of
  Trafo trf → case trf m of
    TrafoE m2 f →
      case f a (T unsafeCoerce) Empty of
        (rb, tt, env2) →
          Result (unsafeCoerce m2)
                 rb
                 (unsafeCoerce env2)
```

The function is named unsafe for a good reason; it effectively switches off the type checker. We believe this implementation of the function *runTrafo* is safe though. We could not find any examples where the use of use *runTrafo* goes wrong. Furthermore, the implementation is operationally identical to the original implementation of *runTrafo*, which is considered type correct according to other compilers than GHC. However, in a paper on typed transformations the use of *unsafeCoerce* feels a bit like cheating. Therefore we also provide, below, a version that is free of both *unsafeCoerce* and lazy pattern matching on existential types. With this solution, however, the *Trafo* type is no longer a real *Arrow*, and hence the special *Arrow* notation cannot be used.

In the type of *runTrafo* above the universal quantification on s is on the outside of the type *Trafo*, whereas the existential quantification on *env2* is inside. Instantiating s with *env2* would be much easier if this was the other way around.

We may move the universal quantification over s inside the quantification over *env2*. This would give us the following type:

```
data Trafo2 m t a b =
                Trafo2 (∀env1 . m env1 → TrafoE2 m t a b env1)
data TrafoE2 m t a b env1 =
  ∃env2 . TrafoE2
           (m env2)
           (∀s . a s → T env2 s → Env t s env1
                 → (b s, T env1 s, Env t s env2)
           )
```

Note that the type variables a and b are now labelled with s, and hence have kind ($*$ → $*$). This is essential because we want to manipulate terms and *Ref*s which are labelled with type s. For example the type of *newRef* which used to be:

```
newSRef :: Trafo m t s (t a s) (Ref a s)
```

now becomes:

```
newSRef2 :: Trafo2 m t (t a) (Ref a)
```

The implementation of *runTrafo* on the new *Trafo2* type is fairly straightforward:

```
runTrafo2 :: Trafo2 m t a b → m () → (∀s . a s) → Result m t b
runTrafo2 trafo m a =
  case trafo of
    Trafo2 trf → case trf m of
      TrafoE2 m2 f →
        let (rb, tt, env2) = f a (T id) Empty
        in  Result m2 rb env2
```

Unfortunately the new data type *Trafo2* is not really an *Arrow*, because the type variables a and b are of kind $* \to *$ instead of $*$. We can however provide an *Arrow*-style interface for programming with the type *Trafo2*, by making it instance of the following class:

```
class Arrow2 arr where
  arr2     :: (∀s . a s → b s) → arr a b
  (>>>>)   :: arr a b → arr b c → arr a c
  first2   :: arr a b → arr (Pair a c) (Pair b c)
  second2  :: arr a b → arr (Pair c a) (Pair c b)
  (****)   :: arr a b → arr a' b' → arr (Pair a a') (Pair b b')
  (&&&&)   :: arr a b → arr a b' → arr a (Pair b b')


data Pair a b s = P (a s, b s)
```

Although the combinators above do not define a real *Arrow*, programming with them is the same as programming with *Arrow*s, except that one cannot use the special *Arrow* syntax [44]. This is unfortunate, because the special syntax make programming with *Arrow*s a lot easier.

5.6 Conclusion

We have shown how to use the Haskell type system and its extensions to perform a fully typed program transformation. Doing so we have used a wide variety of type system concepts: placing existentials precisely at the positions where needed, making things polymorphic where needed, using loop combinators to feed back the result of the computation into the computation inside the scope of an existential, using GADTs to type the environments we construct, scoped type variables, splitting the type labels of the environment into a *use* and a *def* part and thus temporarily decoupling the types of the occurring references and the types associated with the terms in the environment being constructed. We introduced an arrow like style for composing the transformations. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

We think that studying the algorithm and its approaches to the various subproblems is indispensable for anyone who wants to program similar transformation-based algorithms in a strongly typed setting. Some might wonder why the approach taken may be necessary at all, and why not resort to off-line techniques, and they have a point. It is often easier to work in an untyped setting, only to check the generated result afterwards for type correctness. On the other hand one can see the added complexity as a partial correctness proof of the transformation, and as we all know proofs of correct lemmas are superfluous.

We believe that the arrow-based library will turn out to be useful in building programs that transform typed abstract syntax, and that the pattern we have followed in this paper will be followed in many more interesting applications to come.

It is unfortunate that GHC does not support lazy pattern matching on data constructors with existential types. We hope this will be supported in the future. Until then, a user of the library is posed the following dilemma: either have an *unsafeCoerce* in the implementation of *runTrafo*, or use the alternative *Trafo* type, but loose the convenience of the *Arrow*-notation.

5.7 Transformation library

5.7.1 Data types

```
data Equal :: * → * → * where
  Eq :: Equal a a
```

```
data Ref a env where
  Zero ::                  Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)

data Env term use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use
        → Env t use (def', a)

type FinalEnv t usedef = Env t usedef usedef

data Result m t b = ∃s . Result (m s) (b s) (FinalEnv t s)

data T e s = T {unT :: ∀x . Ref x e → Ref x s}

data Unit s = Unit

data Trafo m t s a b =
  Trafo (∀env1 . m env1 → TrafoE m t s a b env1)
data TrafoE m t s a b env1 =
       ∃env2 . TrafoE
                (m env2)
                (a → T env2 s → Env t s env1
                  → (b, T env1 s, Env t s env2)
                )
```

5.7.2 Functions

```
(∼) :: Ref a env → Ref b env → Maybe (Equal a b)

lookupEnv :: Ref a env → Env t s env → t a s

updateEnv :: (t a s → t a s) → Ref a env
          → Env t s env → Env t s env

newSRef :: Trafo Unit t s (t a s) (Ref a s)

extEnv :: t a s → T (e, a) s → Env t s e
       → (Ref a s, T e s, Env t s (e, a))

castSRef :: Ref a env
         → (x          → T env s → Env t s env
         → (Ref a s  ,  T env s  ,  Env t s env))

runTrafo :: (∀s . Trafo m t s a (b s)) → m () → a
         → Result m t b
```

5.7.3 *Arrow* interface

```
arr :: (a → b) → Trafo m t s a b

(>>>) :: Trafo m t s a b → Trafo m t s b c
      → Trafo m t s a c

first :: Trafo m t s a b → Trafo m t s (a, c) (b, c)

second :: Trafo m t s b c → Trafo m t s (d, b) (d, c)

(***) :: Trafo m t s b c → Trafo m t s b' c'
      → Trafo m t s (b, b') (c, c')

(&&&) :: Trafo m t s b c → Trafo m t s b c'
      → Trafo m t s b (c, c')

loop :: Trafo m t s (a, x) (b, x) → Trafo m t s a b
```

5.7.4 *Trafo2*

```
data Trafo2 m t a b =
  Trafo2 (∀env1 . m env1 → TrafoE2 m t a b env1)
data TrafoE2 m t a b env1 =
  ∃env2 . TrafoE2
           (m env2)
           (∀s . a s → T env2 s → Env t s env1
                 → (b s, T env1 s, Env t s env2)
           )

newSRef2 :: Trafo2 m t (t a) (Ref a)

runTrafo2 :: Trafo2 m t a b → m () → (∀s . a s) → Result m t b

class Arrow2 arr where
  arr2    :: (∀s . a s → b s) → arr a b
  (>>>>)  :: arr a b → arr b c → arr a c
  first2  :: arr a b → arr (Pair a c) (Pair b c)
  second2 :: arr a b → arr (Pair c a) (Pair c b)
  (****)  :: arr a b → arr a' b' → arr (Pair a a') (Pair b b')
  (&&&&)  :: arr a b → arr a b' → arr a (Pair b b')

data Pair a b s = P (a s, b s)
```

Chapter 6

Typed Transformations of Typed Grammars: the Left-corner Transform

This is a slightly edited version of a paper originally published as:
Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

Typed Transformations of Typed Grammars the Left-corner Transform

Arthur Baars
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Valencia, Spain

S. Doaitse Swierstra
Department of Computer Science
Utrecht University
Utrecht, The Netherlands

Marcos Viera
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Abstract:

One of the questions which comes up when using embedded domain specific languages is to what extent we can analyze and transform embedded programs, as normally done in more conventional compilers. Special problems arise when the host language is strongly typed, and this host type system is used to type the embedded language. In this paper we describe how we can use a library, which was designed for constructing transformations of typed abstract syntax, in the removal of left recursion from a typed grammar description. The algorithm we describe is the Left-Corner Transform, which is small enough to be fully explained, involved enough to be interesting, and complete enough to serve as a tutorial on how to proceed in similar cases. The described transformation has been successfully used in constructing a compositional and efficient alternative to the standard Haskell *read* function.

6.1 Introduction

In Haskell one can describe how values of a specific data type are to be serialized (i.e. written) and deserialized (i.e. read or parsed). Since data types can be passed as parameter to data type constructors, and definitions can be spread over several modules, the question arises how to dynamically combine separately generated pieces of "reading code" into a function *read* for a composite data type. The standard solution in Haskell, based on a straightforward combination of top-down parsers, has turned out to exhibit exponential reading times. Furthermore, in order to avoid the dynamic construction of left recur-

sive top-down parsers at run-time the input is required to contain many more parentheses than one would expect.

In a recent paper [63] we have presented a solution to this problem; instead of generating code which reads a value of some data type a, the compiler constructs a value of type *Grammar a* which represents the piece of grammar that describes external representations of values of that data type. The striking feature of this grammar type is that it reflects the type of values represented. This is necessary, since from such a value eventually a *read* function of type $String \rightarrow a$ has to be constructed by Haskell library code.

The solution builds upon three, more or less independent, layers (from top to bottom):

1. A template Haskell library which generates the values of type *Grammar a* and library code which combines such values at run-time to form a complete grammar. Out of this combined value the desired read function for the composed data type is constructed, again by library Haskell code. This whole process is described in the aforementioned paper [63].

2. This code calls a library function which removes potential left-recursion from the composed grammar. For this we use the Left-Corner Transform (LCT) [41]. This code, which produces a function of type $Grammar\ a \rightarrow Grammar\ a$, is a fine example of how to express transformations of typed abstract syntax containing references; in the *Grammar a* case these stem from occurrences of non-terminal symbols in the right hand sides of the productions.

3. The LCT and the left-factoring code make use of an intricate Haskell library, which exploits every corner of the Haskell type system and its extensions, such as Generalized Algebraic Data Types, existential and polymorphic types, and lazy evaluation at the type level. The design alternatives and the final design of the library, as it has been made available to the Haskell world, deserved a paper of its own [7].

In this paper we focus on the middle of the above three layers; we start out by presenting an elegant formulation of the LCT in combination with an untyped Haskell implementation, next we introduce the API as implemented by the bottom layer, and we finish by reformulating the untyped version into a typed one using this API.

The LCT [34] is more involved than the direct left recursion removal given in [6], but is also more efficient ($O(n^2)$, where n is the number of terminals and non-terminals in the grammar). Here we will start from an improved version formulated by Robert C. Moore [41], which we present in a more intuitive form. Both his tests, using several large grammars for natural language processing, and our tests [63], using several very large data type descriptions, show that the algorithm performs very well in practice.

What makes this transformation interesting from the typed abstract syntax point of view is that a grammar consists of a collection of grammar rules (one

for each non-terminal) *containing references to other definitions*; we are thus not transforming a tree but a complete binding structure. During this transformation we introduce many new definitions. In the right hand side of these definitions we again use references to such newly introduced symbols. In our setting a transformation must be type preserving and we thus have to ensure that the types of the environment and the references remain consistent, *while being modified*. Previous work on typeful program transformations [11, 45, 6] cannot handle such introductions of new definitions and binders.

We present the algorithm in terms of Haskell code, and thus require Haskell knowledge from the reader. Please keep in mind however that Haskell currently is one of the few general purpose languages in which the problem we describe can be solved at all.

6.2 Left-Corner Transform

In this section we introduce the LCT [34] as a set of construction rules and subsequently give an untyped implementation in Haskell98. Note that, despite being called a transformation, the process is actually constructing a new grammar while inspecting the input grammar. We assume that only the start symbol may derive ϵ.

We say that a symbol X is a *direct left-corner* of a non-terminal A, if there exists a production for A which has the symbol X as its left-most symbol in the right-hand side of that production. We define the *left-corner* relation as the transitive closure of the direct left-corner relation. Note that a non-terminal being left-recursive is equivalent to being a left-corner of itself.

The LCT is defined as the application of three surprisingly simple rules. We use lower-case letters to denote terminal symbols, low-order upper-case letters (A, B, etc.) to denote non-terminals from the grammar and high-order upper-case letters (X, Y, Z) to denote symbols that can either be terminals or non-terminals. Greek symbols denote sequences of terminals and non-terminals.

For a non-terminal A of the original grammar the algorithm constructs new productions for A, and a set of new definitions for non-terminals of the form A_X. A new non-terminal A_X represents that part of A which is still to be recognized after having seen an X. The following rules are applied for each non-terminal until no further results are obtained:

Rule 1 For each production $A \to X\ \beta$ of the original grammar add $A\_X \to \beta$ to the transformed grammar, and add X to the left-corners of A.

Rule 2 For each newly found left-corner X of A:

- **a** If X is a terminal symbol add $A \to X\ A\_X$ to the new grammar.
- **b** If X is a non-terminal then for each original production $X \to X'\ \beta$ add the production $A\_X' \to \beta\ A\_X$ to the new grammar and add X' to the left-corners of A.

As an example consider the grammar:

$$A \rightarrow a\ A \mid B$$
$$B \rightarrow A\ b \mid c$$

Applying rule 1 for the productions of A results in two new productions and two newly encountered left-corners:

$$A\_a \rightarrow A$$
$$A\_B \rightarrow \epsilon \quad sp\_\ leftcorners = [a, B]$$
$$rule2a\_on\ a \Rightarrow$$
$$A \rightarrow a\ A\_a \quad sp\_\ leftcorners = [\cancel{a}, B]$$
$$rule2b\_on\ B \Rightarrow$$
$$A\_A \rightarrow b\ A\_B$$
$$A\_c \rightarrow A\_B\ sp\_\ leftcorners = [\cancel{a}, \_B\_, A, c]$$
$$rule2b\_on\ A \Rightarrow$$
$$A\_a \rightarrow A\ A\_A$$
$$A\_B \rightarrow A\_A\ sp\_\ leftcorners = [\cancel{a}, \_B\_, \_A\_, c]$$
$$rule2a\_on\ c \Rightarrow$$
$$A \rightarrow c\ A\_c \quad sp\_\ leftcorners = [\cancel{a}, \_B\_, \_A\_, \cancel{c}]$$

Since now all left-corners of A have been processed we are done with A. For the non-terminal B the process yields the following new productions:

$$B\_A \rightarrow b \quad \text{-- rule 1}$$
$$B\_c \rightarrow \epsilon \quad \text{-- rule 1}$$
$$B\_a \rightarrow A\ B\_A \quad \text{-- rule 2b, } A$$
$$B\_B \rightarrow B\_A \quad \text{-- rule 2b, } A$$
$$B \rightarrow c\ B\_c \quad \text{-- rule 2a, } c$$
$$B \rightarrow a\ B\_a \quad \text{-- rule 2a, } a$$
$$B\_A \rightarrow b\ B\_B \quad \text{-- rule 2b, } B$$
$$B\_c \rightarrow B\_B \quad \text{-- rule 2b, } B$$

Note that by construction this new grammar is not left-recursive.

6.2.1 The Untyped Left-Corner Transform

Before presenting our typed implementation of the Left-Corner Transform, we first present an untyped version. Grammars are represented by the types:

```
type Grammar_ = Map NT [Prod]
type NT       = String
type Prod     = [Symbol]
type Symbol   = String
isNonterminal = isUpper . head
isTerminal    = isLower . head
```

Thus a *Grammar* is a mapping which associates each non-terminal name with its set of productions. Each production (*Prod*) consists of a sequence of symbols (*Symbol*). So our example grammar can be encoded as:

```
grammar = Map.fromList [("A", [["a", "A"], ["B"]])
                       ,("B", [["A", "b"], ["c"]])]
```

In the transformation process we use the *Control.Monad.State*-monad to store the thus far constructed new grammar. For each non-terminal we traverse the transitive left-corner relation as induced by the productions in depth-first order, while caching the set of thus far encountered left-corner symbols in a list:

```
type LeftCorner = Symbol
type Step_State  = (Grammar_, [LeftCorner])
type Trafo a     = State Step_State a
```

The function *leftcorner* takes a grammar and returns a transformed grammar by running the transformation *rules1*, which yields a value of the monadic type *Trafo*. The state is initialized with an empty grammar and an empty list of encountered left-corner symbols. The final state contains the newly constructed grammar:

```
leftcorner  :: Grammar_ → Grammar_
leftcorner g = fst . snd . runState (rules1 g g) $ (Map.empty, [])
```

For each (*mapM_*) non-terminal (*A*) the function *rules1* visits each (*mapM*) of its productions; each visit results in new productions using *rule2a* and *rule2b*. They are added to the transformed grammar by the function *insert*. The productions resulting from *rule2a* are returned (*ps*), and together (*concat*) from the new productions for the original non-terminal *A*. The left-corners cache is reset when starting with the next non-terminal:

```
rules1 :: Grammar_ → Grammar_ → Trafo ()
rules1 gram nts = mapM_ nt (Map.toList nts)
  where nt (a, prods) =
              do ps ← mapM (rule1 gram a) prods
                 modify  (λ(g, _) → (Map.insert a (concat ps) g, []))
```

For each of the rules given we define a function: *rule2b* generates new productions for non-terminals of the original grammar, and *rule1* and *rule2b* generate productions for non-terminals of the form *A_X*:

```
rule1 :: Grammar_ → NT → Prod → Trafo [Prod]
rule1 grammar a (x : beta) = insert grammar a x beta

rule2a :: NT → Symbol → Prod
rule2a a_b b = [b, a_b]

rule2b :: Grammar_ → NT → NT → Prod → Trafo [Prod]
rule2b grammar a a_b (y : beta) = insert grammar a y (beta ++ [a_b])
```

The function *insert* adds a new production for a non-terminal *A_X* to the grammar: if we have met *A_X* before, the already existing entry is extended and otherwise a new entry for *A_X* is added. In the latter case we apply *rule2* in order to find further left-corner symbols:

```
insert :: Grammar_ → NT → Symbol → Prod → Trafo [Prod]
insert grammar a x p =
  do let a_x = a ++ "_" ++ x
     (gram, lcs) ← get
     if x ∈ lcs then do put (Map.adjust (p:) a_x gram, lcs)
                        return []
                else  do put (Map.insert a_x [p] gram, x : lcs)
                        rule2 grammar a x
```

In *rule2* new productions resulting from applications of *rule2b* are directly inserted into the transformed grammar, whereas the productions resulting from *rule2a* are collected and returned as the result of the *Trafo*-monad. When the newly found left-corner symbol is a terminal *rule2a* is applied, and the resulting new production rule is simply returned. If it is a non-terminal, its corresponding productions are located in the original grammar and *rule2b* is applied to each of them:

```
rule2 :: Grammar_ → NT → Symbol → Trafo [Prod]
rule2 grammar a b
  | isTerminal b = return [rule2a a_b b]
  | otherwise    = do let Just prods = Map.lookup b grammar
                      rs ← mapM (rule2b grammar a a_b) prods
                      return (concat rs)
  where a_b = a ++ "_" ++ b
```

Note that the functions *rule2* and *insert* are mutually recursive. They apply the rules 2a and 2b until no new left-corner symbols are found. The structure of the typed implementation we present in section 6.4 closely resembles the untyped solution above.

6.3 Typed Transformations

The typed version of the LC transform is implemented by using a library (TTTAS[1]) we described in a companion paper [7] to perform typed transformations of typed abstract syntax (in our case typed grammars). In the following subsections we introduce the basic constructs for representing typed abstract syntax and the library interface for manipulating it.

[1] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/TTTAS`.

6.3.1 Typed References and Environments

Pasalic and Linger [45] introduced an encoding *Ref* of typed references pointing into an environment containing values of different type. A *Ref* is actually an index labeled with both the type of the referenced value and the type of the environment (a nested Cartesian product, growing to the right) the value lives in:

```
data Ref a env where
  Zero ::                  Ref a (env', a)
  Suc  :: Ref a env' → Ref a (env', b)
```

The type *Ref* is a generalized algebraic data type [47]. The constructor *Zero* expresses that the first element of the environment has to be of type a. The constructor *Suc* does not care about the type of the first element in the environment (it is polymorphic in b), and remembers a position in the rest of the environment.

We extend this idea such that environments do not contain values of mixed type but *terms* (expressions) describing such values instead; these terms take an extra type parameter describing the environment into which references to other terms occurring in the term may point. In this way we can describe typed terms containing typed references to other terms. As a consequence, an *Env* may be used to represent an environment, consisting of a collection of possibly mutually recursive definitions. The environment stores a heterogeneous list of terms of type *t a use*, which are the right-hand expressions of the definitions. References to elements are represented by indices in the list.

```
data Env term use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use → Env t use (def', a)
```

The type parameter *def* contains all the type labels a of the terms of type *t a use* occurring in the environment. When a term is added to the environment using *Ext*, its type label is included as the first component of *def*. The type *use* describes the types that may be referenced from within terms of type *t a use* using *Ref a use* values. When the types *def* and *use* coincide the type system ensures that the references in the terms do not point to values outside the environment.

The function *lookupEnv* takes a reference and an environment into which the reference points. The occurrence of the two *env*'s in the type of *lookupEnv* guarantees that the lookup will succeed, and that the value found is indeed labeled with the type with which the *Ref* argument was labeled, which is encoded by the two occurrences of a:

```
lookupEnv :: Ref a env → Env t s env → t a s
lookupEnv Zero    (Ext _ t) = t
lookupEnv (Suc r) (Ext ts _) = lookupEnv r ts
```

6.3.2 Transformation Library

The library is based on the type *Trafo*, which represents typed transformation steps. Each transformation step (possibly) extends an implicitly maintained environment *Env*.

```
data Trafo m t s a b
     = Trafo (∀env1 . m env1 → TrafoE m t s env1 a b)
```

The argument m stands for the type of the observable state of the transformation. A *Trafo* takes such a state value, which depends on the environment constructed thus far, as input and yields a new state corresponding to the (possibly extended) environment. The type t is the type of the terms stored in the environment. The type variable s represents the type of the final result, which is passed as the *use* argument in the embedded references. We compose transformations in an arrow style. The arguments a and b are the *Arrow*'s input and output, respectively. The *Arrow* library [30] contains a set of functions for constructing and combining values that are instance of the *Arrow* class. Furthermore there is a convenient notation [44] for programming with *Arrow*s. This notation is inspired by the **do**-notation for *Monad*s. The class *ArrowLoop* is instantiated to be able to construct feedback loops. The TTTAS library includes a combinator, analogous to the *sequence* combinator for *Monad*s, which combines a sequence of transformations into one single large transformation:

```
sequenceA :: [Trafo m t s a b] → Trafo m t s a [b]
```

Each individual transformation maps the input a onto a value b. The combined results b resulting from applying the individual transformations in sequence, are returned as a list $[b]$.

The constructor *Trafo* contains a function which maps a state in the current environment to the actual transformation, represented by the type *TrafoE*. Because the internal details of the type *TrafoE* are of no relevance here, we do not give its definition; we only present its constructors:

```
extEnv      :: m (e, a)          → TrafoE m t s e (t a s) (Ref a s)
castSRef    :: m e → Ref a e → TrafoE m t s e i       (Ref a s)
updateSRef :: m e → Ref a e → (i → t a s → t a s)
                                 → TrafoE m t s e i       (Ref a s)
```

The function *extEnv* builds a *TrafoE* which takes a typed term (of type $t\ a\ s$) as input, adds it to the environment and yields a reference pointing to this value in the final environment (s). The argument of *extEnv* is a state that depends on the extended environment (e, a). Thus, for example, a transformation that extends the environment without keeping any internal state can be implemented:

```
data Unit env = Unit
newSRef :: Trafo Unit t s (t a s) (Ref a s)
newSRef = Trafo (λ_ → extEnv Unit)
```

The function *castSRef* builds a *TrafoE* that returns the reference passed as parameter (in the current environment e) casted to the final environment. The function *updateSRef* builds a *TrafoE* that updates the value pointed by the passed reference. Note that the update function (of type $i \to t\ a\ s \to t\ a\ s$) can use the input of the *Arrow*. The type (*TrafoE m t s e a*) is an instance of the class *Functor*, so the function

$$fmap :: (b \to c) \to TrafoE\ m\ t\ s\ e\ a\ b \to TrafoE\ m\ t\ s\ e\ a\ c$$

lifts a function with type $(b \to c)$ and applies it to the output of the *Arrow*.

When we run a transformation we start with an empty environment and an initial value. Since this argument type is labeled with the final environment, which we do not know yet, is has to be a polymorphic value.

$$runTrafo :: (\forall s\ .\ Trafo\ m\ t\ s\ a\ (b\ s)) \to m\ () \to a \to Result\ m\ t\ b$$

The *Result* contains the final state ($m\ s$), the output value ($b\ s$) and the final environment (*Env t s s*). Since in general we do not know how many new definitions and of which types are introduced by the transformation the result is existential in the final environment s. Despite this existentially, we can enforce the environment to be closed:

$$\textbf{data}\ Result\ m\ t\ b = \exists s\ .\ Result\ (m\ s)\ (b\ s)\ (Env\ t\ s\ s)$$

6.4 The Typed Left-Corner Transform

For a typed version of the LCT we need a typed representation of grammars. A grammar consists of a start symbol, represented as a reference labeled with the type that serves as the witness value of a successful parse, and an *Env*, containing for each non-terminal its list of productions. The actual type *env*, describing the types associated with the non-terminals, is hidden using existential quantification:

$$\begin{aligned}&\textbf{data}\ Grammar\ a = \exists env\ .\ Grammar\ (Ref\ a\ env)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (Env\ Productions\ env\ env)\\ &\textbf{data}\ Productions\ a\ env = PS\ \{unPS :: [Prod\ a\ env]\}\end{aligned}$$

Since in our LCT we want to have easy access to the first symbol of a production we have chosen a representation which facilitates this. Hence the types of the elements in a sequential composition have been chosen a bit different from the usual one [58], such that *Seq* can be chosen to be right associative. The types have been chosen in such a way that if we close the right hand side sequence of symbols with an *End f* element, then this f is a function that accepts the results of the earlier elements (parsing results of the right hand side) as arguments, and builds the parsing result for the left-hand side non-terminal. In our case a

production is a sequence of symbols, and a symbol is either a terminal with a *String* as its witness or a non-terminal (reference):

```
data Symbol :: * → * → * → * where
  Nont :: Ref a env → Symbol a      env
  Term :: String →    Symbol String env
data Prod :: * → * → * → * where
  Seq  :: Symbol b env → Prod (b → a) env → Prod a env
  End  :: a            → Prod  a      env
```

In order to make our grammars resemble normal grammars we introduce some extra operators:

```
infixr 5 `cons`, .*.


cons prods g = Ext g (PS prods)
(.*.) = Seq
```

We now have the machinery at hand to encode our example grammar:

```
_A = Nont      Zero
_B = Nont (Suc Zero)
_a = Term "a"
_b = Term "b"
_c = Term "c"
```

Assume we want the witness type for non-terminal A to be a *String* and for non-terminal B an *Int*:

```
grammar :: Grammar String
grammar = Grammar Zero productions
type Types_nts = (((), Int), String)
productions :: Env Productions Types_nts Types_nts
productions = [_a .*. _A .*. End (++)
              ,_B        .*. End show    ] `cons`
              [_A .*. _b .*. End (λy x → length x + length y)
              ,_c        .*. End (const 1)] `cons` Empty
```

Before delving into the LCT itself we introduce some grammar related functions we will need:

```
append    :: (a → b → c) → Prod a env → Symbol b env → Prod c env
matchSym :: Symbol a env → Symbol b env → Maybe (Equal a b)
mapProd  :: T env1 env2 → Prod a env1 → Prod a env2
```

The function *append* is used in the LCT to build productions of the form $\beta\ X\_A$. Basically it corresponds to the *snoc* operation on lists; we only have to make sure

that all the types match. The function *matchSym* compares two symbols and, if they are equal, returns a witness (*Equal*) of the proof that the types a and b are equal. The function *mapProd* systematically changes all the references to non-terminals occurring in a production. It takes a *Ref*-transformer (*T env1 env2*) to transform references in the environment *env1* to references in the environment *env2*.

```
data T env1 env2 = T {unT :: ∀x . Ref x env1 → Ref x env2}
```

6.4.1 The Typed Transformation

The LCT is applied in turn to each non-terminal (A) of the original grammar. The algorithm performs a depth first search for left-corner symbols. For each left-corner X a new non-terminal A_X is introduced. Additionally a new definition for A itself is added to the transformed grammar.

In the untyped implementation we simply used strings to represent non-terminals. In the typed solution non-terminals are, however, represented as typed references. The first time a production for a non-terminal A_X is generated, we must create a new entry for this non-terminal and remember its position. When the next production for such an A_X is generated we must add it to the already generated productions for this A_X: hence we maintain a finite map from encountered left-corner symbols (X) to references corresponding to the non-terminals (A_X). This finite map again caches the already encountered left-corner symbols:

```
data MapA_X env a env2
   = MapA_X (∀x . Symbol x env → Maybe (Ref (x → a) env2))
```

The type variable *env* comes from the original grammar, and *env2* is the type of the new grammar constructed thus far. The type variable a is the type of the current non-terminal. A left-corner symbol labelled with type x is mapped to a reference to the definitions of the non-terminal A_X in the new grammar, provided it was inserted earlier. The type associated with a non-terminal of the form A_X is $(x \to a)$, i.e. a function that returns the semantics of A, when it is passed the semantics of the symbol X. The empty mapping is defined as:

```
emptyMap :: MapA_X env a env2
emptyMap = MapA_X (const Nothing)
```

We introduce the type-synonym *LCTrafo*, which is the type of the transformation step of the LCT. The type of our terms is *Productions*, and the internal state is a table of type *MapA_X*, containing the encountered left-corner symbols.

```
type LCTrafo env a = Trafo (MapA_X env a) Productions
```

Next we define the function *newNontR* which is a special version of the function *newSRef*, using *MapA_X* as internal state instead of *Unit*. It takes a left-

corner symbol X as argument and yields a *LCTrafo* that introduces a new non-terminal A_X. The input of the *LCTrafo* is the first production (*Productions*) for A_X, and the output is the reference to this newly added non-terminal:

$$\begin{aligned}
&\mathit{newNontR} :: \forall x\ \mathit{env}\ s\ a\ .\ \mathit{Symbol}\ x\ \mathit{env} \\
&\qquad \rightarrow \mathit{LCTrafo}\ \mathit{env}\ a\ s\ (\mathit{Productions}\ (x \rightarrow a)\ s)\ (\mathit{Ref}\ (x \rightarrow a)\ s) \\
&\mathit{newNontR}\ x = \mathit{Trafo}\ \$\ \lambda m \rightarrow \mathit{extEnv}\ (\mathit{extendMap}\ x\ m)
\end{aligned}$$

The symbol X is added to the map of encountered left-corners of A by the function *extendMap*, which records the fact that the newly founded left-corner is the first element of the environment (*Zero*) and the previously added ones have to be shifted one place (*Suc*).

$$\begin{aligned}
&\mathit{extendMap}\ :: \mathit{Symbol}\ x\ \mathit{env} \rightarrow \mathit{MapA_X}\ \mathit{env}\ a\ \mathit{env}' \\
&\qquad \rightarrow \mathit{MapA_X}\ \mathit{env}\ a\ (\mathit{env}', x \rightarrow a) \\
&\mathit{extendMap}\ x\ (\mathit{MapA_X}\ m) = \mathit{MapA_X}\ (\lambda s \rightarrow \mathbf{case}\ \mathit{matchSym}\ s\ x\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{Eq} \rightarrow \mathit{Just}\ \mathit{Zero} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fmap}\ \mathit{Suc}\ (m\ s))
\end{aligned}$$

The index at which the new definition for A is stored is usually different from the index of A in the original grammar. This is a problem as we need to copy parts (the βs in the rules) of the original grammar into the new grammar. The non-terminal references in these parts must be adjusted to the new indexes. To achieve this we first collect all the new references for the non-terminals of the original grammar into a finite map, and then use this map to compute a *Ref*-transformer that is subsequently passed around and used to convert references from the original grammar to corresponding references in the new grammar. The type of this finite map is:

$$\mathbf{data}\ \mathit{Mapping}\ o\ n = \mathit{Mapping}\ (\mathit{Env}\ \mathit{Ref}\ n\ o)$$

The mapping is represented as an *Env*, and contains for each non-terminal of the old grammar, the corresponding reference in the new grammar. The mapping can easily be converted into a *Ref*-transformer:

$$\begin{aligned}
&\mathit{map2trans} :: \mathit{Mapping}\ \mathit{env}\ s \rightarrow T\ \mathit{env}\ s \\
&\mathit{map2trans}\ (\mathit{Mapping}\ \mathit{env}) = T\ (\lambda r \rightarrow (\mathit{lookupEnv}\ r\ \mathit{env}))
\end{aligned}$$

Now all that is left to do is to glue all the pieces defined above together. Each of the following functions corresponds to the untyped version with the same name. We start with the function *insert*:

$$\begin{aligned}
&\mathit{insert} :: \forall \mathit{env}\ s\ a\ x\ .\ \mathit{Env}\ \mathit{Productions}\ \mathit{env}\ \mathit{env} \rightarrow \mathit{Symbol}\ x\ \mathit{env} \\
&\qquad \rightarrow \mathit{LCTrafo}\ \mathit{env}\ a\ s\ (T\ \mathit{env}\ s, \mathit{Prod}\ (x \rightarrow a)\ s)\ (\mathit{Productions}\ a\ s) \\
&\mathit{insert}\ \mathit{old_gram}\ x = \\
&\quad \mathit{Trafo}\ \ (\lambda(\mathit{MapA_X}\ m) \rightarrow \mathbf{case}\ m\ x\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ r\ \ \ \rightarrow \mathit{extendA_X}\ (\mathit{MapA_X}\ m)\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{insNewA_X}\ (\mathit{MapA_X}\ m))
\end{aligned}$$

```
  where
    Trafo insNewA_X = proc (tenv_s, p) →
                        do r ← newNontR x ≺ PS [p]
                           rule2 old_gram x ≺ (tenv_s, r)
```

This function takes the original grammar and a left-corner symbol x as input. It yields a transformation that takes as input a *Ref*-transformer from the original to the new (transformed) grammar and a production for the non-terminal A_X, and stores this production in the transformed grammar. If the symbol x is new ($m\ x$ returns *Nothing*), the production is stored at a new index (using *newNontR*) and the function *rule2* is applied, to continue the depth-first search for left-corners. If we already know that x is a left-corner of a then we obtain an index r to the previously added to the non-terminal A_X, and add the new production at this position. The function *extendA_X* returns the *TrafoE* that performs this update into the environment:

```
extendA_X :: m env1 → Ref (x → a) env1
           → TrafoE m Productions s env1 (T env s, Prod (x → a) s)
                                         (Productions a s)
extendA_X m r = fmap (const $ PS []) $ updateSRef m r addProd
          where addProd (_, p) (PS ps) = PS (p : ps)
```

If in the function *rule2* the left-corner is a terminal symbol then *rule2a* is applied, and the new production rule is returned as *Arrow*-output. In case the left-corner is a non-terminal the corresponding productions are looked up in the original grammar, and *rule2b* is applied to all of them, thus extending the grammar under construction:

```
rule2 :: Env Productions env env → Symbol x env
      → LCTrafo env a s (T env s, Ref (x → a) s) (Productions a s)
rule2 _ (Term a) = proc (_, a_x) → returnA ≺ PS [rule2a a a_x]
rule2 old_gram (Nont b) = case lookupEnv b old_gram of
  PS ps → proc (tenv_s, a_x) →
                do  pss ← sequenceA (map (rule2b old_gram) ps)
                        ≺ (tenv_s, a_x)
                    returnA ≺ PS (concatMap unPS pss)
```

We now define the functions *rule2a*, and *rule2b* that implement the corresponding rules of the LCT. Firstly, *rule2a*, which does not introduce a new non-terminal, but simply provides new productions for the non-terminal (A) under consideration. The implementation of rule 2a is as follows:

```
rule2a :: String → Ref (String → a) s → Prod a s
rule2a a refA_a = Term a . * . Nont refA_a . * . End ($)
```

The function *rule2b* takes the original grammar and a production from the original grammar as arguments, and yields a transformation that takes as input

a *Ref*-transformer and a reference for the non-terminal *A_B*, and constructs a new production which is subsequently inserted. Note that the *Ref*-transformer *tenv_s* is applied to the non-terminal references in *beta* to map them on the corresponding references in the new grammar.

```
rule2b :: Env Productions env env → Prod b env
       → LCTrafo env a s (T env s, Ref (b → a) s) (Productions a s)
rule2b old_gram (Seq x beta)
  = proc (tenv_s, a_b) →
      insert old_gram x ≺ (tenv_s
                          , append (flip (.))
                                   (mapProd tenv_s beta)
                                   (Nont a_b)
                          )
```

The function *rule1* is almost identical to *rule2b*; the only difference is that it deals with direct left-corners and hence does not involve a "parent" non-terminal *A_B*.

```
rule1 :: Env Productions env env → Prod a env
      → LCTrafo env a s (T env s) (Productions a s)
rule1 old_gram (Seq x beta)
  = proc tenv_s → insert old_gram x ≺ (tenv_s, mapProd tenv_s beta)
```

The function *rules1* is defined by induction over the original grammar (i.e. it iterates over the non-terminals) with the second parameter as the induction parameter. It is polymorphically recursive: the type variable *env′* changes during induction, starting with the type of the original grammar (i.e. *env*) and ending with the type of the empty grammar (). The first argument is a copy of the original grammar which is needed for looking up the productions of the original non-terminals:

```
rules1 :: Env Productions env env → Env Productions env env′
       → Trafo Unit Productions s (T env s) (Mapping env′ s)
rules1 _ Empty      = proc _ → returnA ≺ Mapping Empty
rules1 old_gram (Ext ps (PS prods))
  = proc tenv_s →
      do p           ← initMap nt          ≺ tenv_s
         r           ← newSRef             ≺ p
         Mapping e ← rules1 old_gram ps ≺ tenv_s
         returnA ≺ Mapping (Ext e r)
       where
         nt = proc tenv_s →
                do pss ← sequenceA (map (rule1 old_gram) prods)
                       ≺ tenv_s
                   returnA ≺ PS (concatMap unPS pss)
```

The result of *rules1* is the complete transformation represented as a value of type *Trafo*. At the top-level the transformation does not use any state, hence the type *Unit*. When dealing with one non-terminal (*nt*), *rule1* is applied for each of its productions and the new productions are collected to be inserted in the new grammar. The function *initMap* initializes the state information of the transformation *nt* with an empty table of encountered left-corners.

```
initMap :: LCTrafo env a s c d → Trafo Unit Productions s c d
initMap (Trafo st) = Trafo (λ_ → case st emptyMap of
                                   TrafoE _ f → TrafoE Unit f)
```

As input the transformation returned by *rules1* needs a *Ref*-transformer to remap non-terminals of the old grammar to the new grammar. During the transformation *rules1* inserts the new definitions for non-terminals of the original grammar, and remembers the new locations for these non-terminals in a *Mapping*. This *Mapping* can be converted into the required *Ref*-transformer, which must be fed-back as the *Arrow*-input. This feed-back loop is made in the function *leftcorner* using **mdo**-notation:

```
leftcorner :: ∀a . Grammar a → Grammar a
leftcorner (Grammar start productions)
      = case runTrafo lctrafo Unit ⊥ of
        Result _ (T tt) gram → Grammar (tt start) gram
  where
    lctrafo = proc _ → mdo
      let tenv_s = map2trans menv_s
      menv_s ← (rules1 productions productions) ≺ tenv_s
      returnA ≺ tenv_s
```

The resulting transformation is run using $\bot$ as input; this is perfectly safe as it does not use the input at all: the result is a new start symbol and the transformed production rules, which are combined to form the new grammar.

6.5 Conclusions

We have shown how complicated transformations can be done at run-time, while having been partially verified statically by the type system. Doing so we have used a wide variety of type system concepts, like GADTs and existential and polymorphic types, which cannot be found together in other general purpose languages than Haskell. This allows us to use techniques which are typical of dependently typed systems while maintaining a complete separation between types and values. Besides this we make use of lazy evaluation in order to get computed information to the right places to be used.

Implementing transformations like the left-corner transform implies the introduction of new references to a collection of possibly mutually recursive definitions. Previous work on typeful transformations of embedded DSLs represented

as typed abstract syntax [11, 6, 23] does not deal with such complexity. Thus, as far as we know, this is the first description of run-time typed transformations which modify references into an abstract syntax represented as a graph instead of a tree.

We have shown how the untyped version of a transformation can be transformed into a typed version; after studying this example the implementation of similar transformations, using the TTTAS library, should be relatively straightforward. Despite the fact that this transformation is rather systematic, it remains a subject of future research to see how such transformations can be done automatically.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Summary in *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[3] Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

[4] Arthur I. Baars and S. Doaitse Swierstra. Syntax macros. `http://www.cs.uu.nl/groups/ST/Center/SyntaxMacros`.

[5] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

[6] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

[7] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

[8] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

[9] Robert D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(4):85–94, March 1993.

[10] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.

[11] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM'03*, 2003.

[12] James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report TR2003-1901, Cornell University, 2003. `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901`.

[13] Olaf Chitil. Pretty printing with lazy dequeues. *ACM Transactions on Programming Languages and Systems*, 2005.

[14] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, September 2000.

[15] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 62–73. Springer-Verlag, 1999.

[16] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Utrecht University, 2001.

[17] Dave Clarke and Andres Löh. Generic Haskell, specifically. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.

[18] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[19] Levent Erkök and John Launchbury. A recursive do for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 29–37. ACM Press, 2002.

[20] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming, First International Spring School*, volume 925 of *LNCS*, pages 1–23, Båstad, Sweden, May 1995. Springer-Verlag.

[21] Glasgow Haskell Compiler. Available from: `http://www.haskell.org/ghc/`.

[22] Andy Gill and Simon Marlow. *Happy – The Parser Generator for Haskell*, 1995. University of Glasgow.

[23] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in Haskell. *Electron. Notes Theor. Comput. Sci.*, 174(7):23–39, 2007.

[24] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In M.-C. Silaghi and M. Zanker, editors, *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[25] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[26] Helium compiler. Available from: `http://www.cs.uu.nl/wiki/Helium`.

[27] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. Selected papers of the Fourth European Symposium on Programming (Rennes, 1992).

[28] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[29] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.

[30] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[31] John Hughes and Doaitse Swierstra. Polish parsers, step by step. In *International Conference on Functional Programming*, volume 8, pages 239–248, Uppsala, Sweden, August 2003. ACM Sigplan.

[32] Hugs. `http://www.haskell.org/hugs/`.

[33] G. Hutton and H.J.M. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1998.

[34] M. Johnson. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *COLING-ACL '98, Montreal, Quebec, Canada*, pages 619–623. Association for Computational Linguistics, 1998.

[35] S. C. Johnson. YACC: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353 – 387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[36] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

[37] B. M. Leavenworth. Syntax macros and extended translation. *CACM*, 9(11):790–793, 1966.

[38] Daan Leijen. Parsec, a fast combinator parser. `http://www.cs.uu.nl/~daan/parsec.html`, November 2001.

[39] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[40] Xavier Leroy and Michael Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523, pages 406–426, Berlin, Heidelberg, New York, 1991. Springer-Verlag.

[41] R. Moore. Removing left recursion from context-free grammars, 2000.

[42] Objective Caml. Available from: `http://caml.inria.fr`.

[43] Ross Paterson. Hackagedb: Control.Applicative.Permutation. `http://hackage.haskell.org/package/action-permutations`.

[44] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[45] Emir Pašalić and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering*, volume 3286 of *LNCS*, pages 136–167, 2004.

[46] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. `http://www.haskell.org/report`.

[47] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *ICFP'06, SIGPLAN Not.*, 41(9):50–61, 2006.

[48] Simon L. Peyton Jones. Haskell pretty-printing library. `http://www.haskell.org/libraries/#prettyprinting`.

[49] Simon L. Peyton Jones. Parsing distfix operators. *CACM*, 29:118–122, 1986.

[50] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages, 10th International Workshop, IFL'98*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1999.

[51] Rinus Plasmeijer and Marco van Eekelen. Clean Language Report, version 2.0. `http://www.cs.kun.nl/~clean/`, 2002.

[52] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.

[53] Joost Rommes. Syntax macros: Attribute redefinitions. Master's thesis, Utrecht University, the Netherlands, April 2003. INF/SCR-2003-31.

[54] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM Press, 2002.

[55] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.

[56] Guy L. Steele Jr. Growing a language. *Journal of Higher-Order and Symbolic Computation*, 12:221–236, 1999.

[57] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[58] Doaitse Swierstra and Luc Duponcheel. Deterministic, error correcting combinator parsers. In *Advanced Functional Programming, Second International Spring School*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[59] S. D. Swierstra. Combinator parsers: From toys to tools. In Graham Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[60] S. D. Swierstra, P.R. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In S. D. Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.

[61] Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.

[62] Utrecht Haskell Compiler. `http://www.cs.uu.nl/wiki/UHC`.

[63] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell do you read me? Constructing and composing efficient top-down parsers at runtime. In A. Gill, editor, *Haskell Symposium*. ACM, 2008.

[64] Philip Wadler. How to replace failure with a list of successes. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, 1985.

[65] Philip Wadler. Theorems for free. In Mac Queen, editor, *4'th International Conference on Functional Programming and Computer Architecture*, pages 347–359, London, September 1989. Addison-Wesley.

[66] Philips Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The fun of programming*, pages 223–244. Palgrave Macmillan, 2003.

[67] Stephanie Weirich. Type-safe cast. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 58–67. ACM press, 2000.

[68] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 289–300. ACM press, 1998.

Acknowledgements

I would like to thank everyone who has helped me on the path to this thesis. It has taken much longer than initially expected, but now it is finally done.

A big thanks goes to my supervisor, Doaitse Swierstra, who has supported me throughout my PhD studies. I really enjoyed working with you, and I'll miss the late afternoon discussions and puzzles on the intricacies of the Haskell type system. Thanks also for relentlessly pushing me to finish my thesis. It must have been frustrating for you at times, and I cannot thank you enough for this.

Thanks as well for my co-authors Andres and Marcos, it was a pleasure writing those papers together with you. Andres, thanks also for the `lhs2TeX` tool that made formatting all the Haskell code in this thesis really easy.

My fellow PhD students: Andres, Bastiaan, Karina, Ade, Rui, Daan, Martijn and Eelco also deserve thanks, for sharing ideas, reviewing and commenting draft versions of my papers and in general for providing a great working atmosphere. Thanks as well to Jan de Wit for bringing the work of Leibniz to my attention, and to Jurriaan for his ever welcome distractions.

I would also like to thank all teachers and researchers of the Software Technology group. In particular, Atze for his work on the Utrecht Haskell Compiler. It think we both benefitted from our cooperation.

Since two years I have been working at the Instituto Tecnológico de Informática (ITI) in Valencia. Tanja, thanks for providing me the opportunity to embark on this spanish adventure. You and Santiago have been a great help, making me feel at home in Spain. I would also like to thank the members of the SQuaC group at the ITI: Maximiliano, Jordi, Ignacio, and Veronica. I am also grateful to the ITI, and to Miguel Fuster in particular for supporting my trip to the LDTA conference.

I would like to thank the members of the examination committee, Roland Backhouse, Lambert Meertens, Rinus Plasmeijer, Johan Jeuring, and Vincent van Oostrom, for reading and approving my thesis and providing me with useful comments to improve this work.

Finally I am grateful to my family. Especially my mother, my sister, my father, and Jan who has always been like a father to me. Thank you for your love and support; I could not have done it without you.

Samenvatting

Om te kunnen automatiseren is het belangrijk om de kennis van experts van een bepaald vakgebied om te zetten in een vorm die geschikt is voor verwerking door computers. Ieder vakgebied heeft zijn eigen vaktermen en manieren om informatie uit te drukken, bijvoorbeeld middels getekende diagrammen, formules, of speciale talen. In het laatste geval spreken we van een domein-specifieke taal.

Het ontwerpen van een domein-specifieke taal mag eenvoudig lijken op het eerste gezicht. De eerste stappen zijn vaak erg succesvol, maken het automatiseringproces eenvoudiger en leveren kostenbesparingen op. Echter blijft het hier meestal niet bij. Gedreven door het initiele succes, krijgen we ideeën voor verbeteringen en worden er meer en meer uitbreidingen gemaakt zonder na te denken over de consistentie van het geheel. Uiteindelijk groeit iets wat elegant en eenvoudig begon uit tot een groot inconsistent en onwerkbaar "monster".

Het vanaf de grond ontwerpen van een complete en consistente taal en het implementeren van een compiler (vertaler) voor deze taal is geen eenvoudige taak, en dus rijst de vraag hoe dit vereenvoudigd kan worden. Een oplossing is om de domein-specifieke taal in te bedden in een al bestaande programmeertaal. Deze aanpak heeft vele voordelen. Men hoeft geen nieuwe compiler (vertaler) te implementeren. De geavanceerde eigenschappen, zoals het type-systeem en de abstractie-mechanismen, van de al bestaande programmeertaal kunnen hergebruikt worden. Naast de compiler en eigenschappen van de gastheer-programmeertaal kunnen ook de grafische ontwikkelomgeving en andere gereedschappen hergebruikt worden. Ten slotte bestaat de mogelijkheid om diverse ingebedde talen te combineren in een programma.

Een veel gebruikte techniek in de Haskell-gemeenschap is het inbedden van domein specifieke talen middels een "combinator library". De programmeertaal Haskell wordt gekenmerkt door een rijk type-systeem dat geschikt is om de type systemen van vele domein specifieke talen te simuleren. Daarnaast bieden de flexibele notatievormen zoals gebruikersgedefinieerde operatoren, type klassen en monad- en lijstcomprehensie de mogelijkheid om de taalconstructies van de ingebedde domein-specifieke taal te specificeren.

De denotationele semantiek van een op combinators gebaseerde ingebedde taal wordt normaal gesproken direct uitgedrukt in de vorm van Haskell functies. Deze impliciete representatie maakt het onmogelijk om conventionele vertalerbouwtechnieken, toe te passen. Een logische vervolgstap is om eerst een expliciete intermedaire structuur te construeren, welke vervolgens geanalyseerd,

getransformeerd en geoptimaliseerd kan worden zoals dit gebeurd in een normale vertaler. Hoe dit effectief gedaan kan worden in een sterk getypeerde gastheertaal zoals Haskell is het onderwerp van dit proefschrift: *Ingebedde Vertalers (Embedded Compilers)*.

Dit proefschrift bestaat uit de vijf hieronder genoemde publicaties. De eerste presenteert een implementatie van "permutation phrases", als uitbreiding op een ingebedde domein specifieke taal voor het beschrijven van ontleders. De laatste vier vormen een serie en behandelen het *Ingebedde Vertalers* thema dat hierboven is uiteengezet.

- Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Functional pearl: Parsing permutation phrases. *Journal of Functional Programming*, 14(06):635–646, 2004.

- Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 157–166. ACM Press, 2002.

- Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self-inspecting code. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79, New York, NY, USA, 2004. ACM Press.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

- Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. In *Proceedings of the 9th Workshop on Language Descriptions Tools and Applications*, ENTCS, pages 18–33, 2009.

Curriculum Vitae

Arthur Iwan Baars

1 November 1977
Geboren te Gouda.

augustus 1990 - juni 1996
VWO aan het Minkema College te Woerden.
Diploma behaald op 12 juni 1996.

september 1996 - november 2000
Studie Informatica aan de Universiteit Utrecht.
Doctoraal diploma behaald op 30 november 2000.

december 2000 - november 2004
Assistent in Opleiding aan het Informatica Instituut van de Universiteit Utrecht.

september 2007 - september 2009
Onderzoeker aan het Instituto Tecnológico de Informática te Valencia, Spanje.

november 2009 - heden
Onderzoeker aan het Departamento de Sistemas Informáticos y Computación van de Technische Universiteit van Valencia, Spanje.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics.* Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of

Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty

of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty

of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25