Julio Mariño (Ed).

Functional and (Constraint) Logic Programming

19th International Workshop, WFLP 2010
Madrid, Spain, January 17th, 2010.
Informal Proceedings

Preface

This report contains preliminary versions of the papers presented at the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010), held at Madrid, Spain, on January 17th, 2010 as part of the ACM-SIGPLAN Principles of Programming Languages event, (POPL 2010). Final versions of a selection of these papers will appear as a forthcoming volume in the Lecture Notes in Computer Science, Springer. The aim of the WFLP workshop is to bring together researchers interested in functional programming, (constraint) logic programming, as well as the integration of the two paradigms. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications and combinations of high-level, declarative programming languages and related areas. Previous meetings were: WFLP 2009 (Brasilia, Brazil), WFLP 2008 (Siena, Italy), WFLP 2007 (Paris, France), WFLP 2006 (Madrid, Spain), WCFLP 2005 (Tallinn, Estonia), WFLP 2004 (Aachen, Germany), WFLP 2003 (Valencia, Spain), WFLP 2002 (Grado, Italy), WFLP 2001 (Kiel, Germany), WFLP 2000 (Benicassim, Spain), WFLP'99 (Grenoble, France), WFLP'98 (Bad Honnef, Germany), WFLP'97 (Schwarzenberg, Germany), WFLP'96 (Marburg, Germany), WFLP'95 (Schwarzenberg, Germany), WFLP'94 (Schwarzenberg, Germany), WFLP'93 (Rattenberg, Germany), and WFLP'92 (Karlsruhe, Germany). The Program Committee of WFLP 2010 selected 12 papers for presentation at the workshop out of 15 submissions, after a process in which most papers received four reviews and the subsequent discussion. In addition to the reviewed papers, the scientific program includes an invited lecture by Mariangiola Dezani-Ciancaglini, from the Universitá di Torino. I would like to thank her specially for having accepted our invitation. I would also like to thank all the members of the Program Committee and the external referees for their careful work in the reviewing process. Last, but not least, we want to express our gratitude to the Facultad de Informática of Universidad Politécnica de Madrid for printing these proceedings and all the people involved in the local organization of the POPL 2010 week.

Madrid, January 2010

Julio Mariño Carballo
Program Chair
WFLP 2010

Organization

WFLP 2010 was organized by the Babel research group at Universidad Politécnica de Madrid, in close collaboration with the organizers of the POPL 2010 week. Ana María Fernández Soriano was in charge of the administrative duties. Emilio Jesús Gallego Arias maintained the web page and composed these preliminary proceedings.

Program Chair

Julio Mariño Carballo	Universidad Politécnica de Madrid, Spain

Program Committee

María Alpuente	Universidad Politécnica de Valencia, Spain
Sergio Antoy	Portland State University, USA
Bernd Brassel	Christian-Albrechts-Universität zu Kiel, Germany
Olaf Chitil	University of Kent, UK
Rachid Echahed	Institut IMAG – Laboratoire Leibniz, Grenoble, France
Santiago Escobar	Universidad Politécnica de Valencia, Spain
Moreno Falaschi	University of Siena, Italy
Murdoch James Gabbay	Heriot-Watt University, UK
María García de la Banda	Monash University, Australia
Víctor Gulias	LambdaStream S.L., Corunna, Spain
Michael Hanus	Christian-Albrechts-Universität zu Kiel, Germany
Herbert Kuchen	Westfälische Wilhelms-Universität Münster, Germany
Francisco José López Fraguas	Universidad Complutense de Madrid, Spain
James B. Lipton	Wesleyan University, USA
Juan José Moreno Navarro	Ministry of Science & Innovation, Spain
Mircea Marin	University of Tsukuba, Japan
Brigitte Pientka	McGill University, Canada

Referees

In addition to the members of the Program Committee, the following external referees contributed to the paper reviewing process:

Sponsoring Institutions

Universidad Politécnica de Madrid, Spain.
IMDEA SW, Madrid, Spain.

Table of Contents

19th Workshop on Functional and (Constraint) Logic Programming

Invited Talk: Sessions and Session Types

Mariangiola Dezani-Ciancaglini

Universitá di Torino
dezani@di.unito.it

Abstract. Sessions are a common and widespread mechanism of interaction in distributed architectures. The processes willing to interact establish a connection on a shared public channel. In this connection they agree on some private channel on which to have a conversation, dubbed session. The conversation follows a given protocol which describes the kind and order of the messages exchanged on the private channel. The messages exchanged during a session may be synchronisation signals, basic values (e.g., integers, booleans, strings), names of public channels (those used to start sessions), or even names of private channels of already started sessions. In the last case one speaks of delegation since by sending to some other process the private channel of a session, the process delegates the receiver to continue that session.
Session types describe the sequences of messages exchanged on a private session channel and their possible branching based on labels: they prevent communication mismatches and session deadlocks. The aim of this talk is to give an overview of sessions/session types and some hints on new proposals for incorporating secure information flow requirements within session types.

Transforming Functional Logic Programs into Monadic Functional Programs

Bernd Braßel, Sebastian Fischer, Michael Hanus, Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|sebf|mh|fre}@informatik.uni-kiel.de

Abstract. We present a high-level transformation scheme to translate lazy functional logic programs into pure Haskell programs. This transformation is based on a recent proposal to efficiently implement lazy non-deterministic computations in Haskell into monadic style. We build on this work and define a systematic method to transform lazy functional logic programs into monadic programs with explicit sharing. This results in a transformation scheme which produces high-level and flexible target code. For instance, the target code is parametric w.r.t. the concrete evaluation monad. Thus, different monad instances could, for example, define different search strategies (e.g., depth-first, breadth-first, parallel). We formally describe the basic compilation scheme and some useful extensions.

1 Introduction

Functional logic languages (see [10] for a recent survey) integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. In particular, modern languages of this kind, such as Curry [13] or $\mathcal{TOY}$ [16], amalgamate the concepts of demand-driven evaluation from functional programming with non-deterministic evaluation from logic programming. This combination is not only desirable to obtain efficient evaluation but it has also a positive effect on the search space, i.e., lazy evaluation on non-deterministic programs yields smaller search spaces due to its demand-driven exploration of the search space (compare [10]).

Although the combination of such features is quite useful for high-level application programming, their implementation is challenging. Many older implementations (e.g., [11,16,18]) are based on Prolog's depth-first backtracking strategy to explore the search space. Since this strategy leads to operational incompleteness and reduces the potential of modern architectures for parallelism, more recent implementations of functional logic languages offer more flexible search strategies (e.g., [5,7]). In order to avoid separate implementations for different strategies, it would be desirable to specify the search strategy (e.g., depth-first, breadth-first, parallel) as a parameter of the implementation. A first step towards such an implementation has been done in [8] where a Haskell library for non-deterministic programming w.r.t. different strategies is proposed. In this paper, we use this idea to compile functional logic programs into pure Haskell programs that are

parameterized such that the generated code works with different run-time systems (various search strategies, call-time/run-time choice, etc).

Before presenting our compilation scheme, we review the features of functional logic programming that we are going to implement as well as the language Curry that we use for concrete examples. From a syntactic point of view, a Curry program is a functional program (with a Haskell-like syntax [19]) extended by non-deterministic rules and free (logic) variables in defining rules. For the sake of simplicity, we do not consider free variables, since it has been shown that they can be replaced by non-deterministic rules [4]. Actually, we use a kernel language, also called *overlapping inductively sequential programs* [2], which are functional programs extended by a (don't know) non-deterministic choice operator "`?`". This is not a loss of generality, since (1) any functional logic program (with extra variables and conditional, overlapping rules) can be transformed into an overlapping inductively sequential program [1], and (2) narrowing computations in inductively sequential programs with free variables are equivalent to computations in overlapping inductively sequential programs without free variables [4, Th. 2].

A *functional logic program* consists of the definition of functions and data types on which the functions operate. For instance, the data types of Booleans and polymorphic lists are defined as

```
data Bool   = True | False
data List a = Nil  | Cons a (List a)
```

Concatenation of lists and an operation that duplicates a list can be defined as:

```
append :: List a  → List a  → List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

dup :: List a  → List a
dup xs = append xs xs
```

Note that functional logic programs require a strict separation between *constructor symbols* (like `True`, `False`, `Nil`, or `Cons`) and *defined functions* or *operations* (like `append` or `dup`). In contrast to general term rewriting, the formal parameters in a rule defining a function contain only variables and constructor symbols. This restriction also holds for pure functional or logic programs and is important to provide efficient evaluation strategies (see [10] for more details).

Logic programming aspects become relevant when considering *non-deterministic operations*, i.e., operations that yield more than one result. For this purpose, there is a distinguished choice operator "`?`" which returns non-deterministically one of its arguments. For instance, the following operation returns a one-element list containing either `True` or `False`:

```
oneBool :: List Bool
oneBool = Cons (True ? False) Nil
```

Now, consider an expression that duplicates the result of the previous operation:

```
main = dup oneBool
```

What are possible values of `main`? One could argue (in pure term rewriting) that "`Cons True (Cons False Nil)`" is a value of `main` by deriving it

to "`append oneBool oneBool`", and then the first argument to "`Cons True Nil`" and the second to "`Cons False Nil`" (this semantics is called *run-time choice* [14]). But this result is not desired as the operation `dup` is intended to *duplicate* a given list (rather than return the concatenation of two different lists). In order to obtain this behavior, González-Moreno et al. [9] proposed a rewriting logic as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [14] where values of the arguments of an operation are determined before the operation is evaluated. Note that this does not necessarily mean that operations are evaluated eagerly. One can still evaluate operations lazily provided that actual arguments passed to operations are shared. For instance, the two occurrences of argument "`xs`" of operation `dup` are shared, i.e., the actual argument `oneBool` is evaluated to the same value on both positions. Thus, "`Cons True (Cons True Nil)`" and "`Cons False (Cons False Nil)`" are the only values of `main`, as intended. Detailed descriptions of this operational semantics can be found in [9,10].

In functional logic programs, non-deterministic operations can occur in any level of the program, in particular, inside nested structures, as shown in operation `oneBool` above. This makes the transformation of such programs into pure functional programs non-trivial. For instance, the traditional functional representation of non-deterministic computations as "lists of successes" [20] is not easily applicable, as one might expect, due to the arbitrary nesting of non-deterministic operations. In the following section we review a recent solution to this problem [8].

2 Lazy, Monadic Non-determinism

In the previous section, we have introduced Curry which combines demand driven with non-deterministic evaluation. While both features can be easily expressed separately in a functional language, their combination is non-trivial. In this section we summarize previous work [8] that shows why.

Demand-driven evaluation is built into *lazy* execution mechanisms of the functional language Haskell. Laziness combines *non-strict* execution (expressions are evaluated only if needed) with *sharing* (expressions are evaluated at most once). Non-deterministic evaluation can be simulated in Haskell via lists or, more generally, non-determinism monads, i.e., instances of the `MonadPlus` type class. The `MonadPlus` type class specifies the following overloaded operations to express non-deterministic computations.[1]

```
mzero   :: MonadPlus m ⇒ m a
return  :: MonadPlus m ⇒ a → m a
mplus   :: MonadPlus m ⇒ m a → m a → m a
(>>=)   :: MonadPlus m ⇒ m a → (a → m b) → m b
```

[1] In fact, `return` and "`>>=`" have more general types because they are not only available in non-determinism monads but in arbitrary instances of the `Monad` type class.

`mzero` denotes a failing computation, i.e., one without results, `return` creates a deterministic computation, i.e., one with a single result, `mplus` creates a non-deterministic choice between the results of the two argument computations, and "`>>=`" applies a non-deterministic function to every result of a non-deterministic computation. For lists, `mzero` is the empty list, `return` creates a singleton list, `mplus` is list concatenation, and `>>=` (pronounced 'bind') can be implemented by mapping the given function over the given list and concatenating the results.

The Curry expression `(True ? False)` can be expressed monadically:

```
trueOrFalse :: MonadPlus ⇒ m Bool
trueOrFalse = mplus (return True) (return False)
```

The constructors `True` and `False` are wrapped with `return` and the resulting computations are combined with `mplus` which replaces Curry's non-deterministic choice operator "`?`". When evaluated in the list monad, `trueOrFalse` yields `[True,False]` which can be verified in a Haskell environment:

```
ghci> trueOrFalse :: [Bool]
[True,False]
```

However, different implementations of the `MonadPlus` interface can be used, e.g., to influence the search strategy. If we use the `Maybe` monad rather than the list monad, we just get one result in depth-first order:

```
ghci> trueOrFalse :: Maybe Bool
Just True
```

The overloading of `trueOrFalse` allows us to execute it using different types. Programs that are compiled with our transformation scheme are also overloaded and can be executed by different monad instances.

We motivate the monadic implementation that we use in our transformation by a sequence of ideas that leads to the final design. A simple idea to translate the Curry operation `oneBool` into monadic Haskell is to reuse the existing Curry data types and *bind* non-deterministic arguments of their constructors:

```
oneBoolM1 :: MonadPlus m ⇒ m (List Bool)
oneBoolM1 = trueOrFalse >>= λb → return (Cons b Nil)
```

We feed the result of function `trueOrFalse` above into a singleton list using the "`>>=`" operator. Like the corresponding Curry operation, `oneBoolM1` yields a singleton list that contains either `True` or `False` non-deterministically:

```
ghci> oneBoolM1 :: [List Bool]
[Cons True Nil, Cons False Nil]
```

However, there is a subtle difference w.r.t. laziness. In Curry, `oneBool` yields the head-normal form of its result without executing the non-deterministic choice inside the list, whereas `oneBoolM1` first executes the non-deterministic choice between `True` and `False` and yields a list with a deterministic first element in each non-deterministic branch of the computation. Whereas in Curry, non-determinism can be nested inside data structures, the monadic non-determinism presented so far cannot.

To overcome this limitation, we can use data types with nested non-deterministic components. Nested monadic lists can be defined by wrapping

each constructor argument with an additional type parameter "m" that represents a non-determinism monad:

```
data MList m a = MNil | MCons (m a) (m (MList m a))
```

The additional "m"s around the arguments of MCons allow to wrap non-deterministic computations inside lists. Here is a different translation of the Curry operation oneBool into monadic Haskell:

```
oneBoolM :: MonadPlus m ⇒ m (MList m Bool)
oneBoolM = return (MCons trueOrFalse (return MNil))
```

This function *deterministically* yields a singleton list with an element that is a non-deterministic choice:

```
ghci> oneBoolM :: [MList [] Bool]
[MCons [True,False] [MNil]]
```

This translation of the Curry operation is more accurate w.r.t. laziness because the MCons constructor can be matched without distributing the non-determinism in its first argument. In order to print such nested non-deterministic data in the usual way, we need to distribute non-determinism to the top level [8].

Now that we have changed the list data type in order to support nested non-determinism, we need to re-implement the list functions defined in Section 1. The monadic variant of the dup function takes a monadic list as argument and yields a monadic list as result:

```
dupM1 :: MonadPlus m ⇒ m (MList m a) → m (MList m a)
dupM1 xs = appendM xs xs
```

Similarly, the monadic variant of append takes two monadic lists and yields one.

```
appendM :: MonadPlus m ⇒ m (MList m a) → m (MList m a)
                       → m (MList m a)
appendM l ys =
 l >>= λl' → case l' of
               MNil → ys
               MCons x xs → return (MCons x (appendM xs ys))
```

This definition resembles the Curry definition of append but additionally handles the monadic parts inside and around lists. In order to match on the first argument "l" of appendM, we *bind* one of its non-deterministic head-normal forms to the variable "l'". Depending on the value of "l'", appendM yields either the second argument "ys" or a list that contains the first element "x" of "l'" and the result of a recursive call (which can both be non-deterministic).

Although such a translation with nested monadic data accurately models non-strictness, it does not ensure sharing of deterministic results. The definition of dupM1 given above uses the argument list "xs" twice and hence, the value of "xs" is shared via Haskell's built-in laziness. However, in dupM1 the variable "xs" denotes a *non-deterministic computation* that yields a list and the built-in sharing does not ensure that both occurrences of "xs" in dupM1 denote the same *deterministic result* of this computation. Hence, the presented encoding of nested monadic data implements run-time choice instead of call-time choice:

```
ghci> dupM1 oneBoolM :: [MList [] Bool]
```

```
[MCons [True,False] [MCons [True,False] [MNil]]]
```

When distributed to the top-level, the non-determinism in the list elements leads to lists with different elements because the information that both elements originate from the same expression is lost.

The conflict between non-strictness and sharing in presence of monadic non-determinism has been resolved recently using an additional monadic combinator for explicit sharing [8]:

```
share :: (Sharing m, Shareable m a) ⇒ m a → m (m a)
```

The type class context of `share` specifies that "`m`" (referring to a non-determinism monad throughout this paper) and the type denoted by "`a`" support explicit sharing. Using `share`, the Curry function `dup` can be translated as follows:

```
dupM :: (MonadPlus m, Sharing m, Shareable m a) ⇒
        m (MList m a) → m (MList m a)
dupM xs = share xs >>= λxs → appendM xs xs
```

The result of `share xs` is a monadic computation that yields itself a monadic computation which is similar to "`xs`" but denotes the same deterministic result when used repeatedly. Hence, the argument "`xs`" to `appendM` (which intentionally shadows the original argument "`xs`" of `dupM`) denotes the same deterministic list in both argument positions of `appendM` which ensures call-time choice. When executing "`dupM oneBoolM`" in a non-determinism monad with explicit sharing, the resulting lists do not contain different elements.

The library that implements `share`, and that we use to execute transformed functional logic programs, is available online[2]. The implementation ideas, the operation that allows to observe results of computations with explicit sharing, as well as equational laws that allow to reason about such computations are not in the scope of this paper but described elsewhere [8].

3 Transforming Functional Logic Programs

In this section we formally define the transformation of functional logic programs into monadic functional programs, i.e., pure Haskell programs. In order to simplify the transformation scheme, we consider functional logic programs in flat form as a starting point of our transformation. Flat programs are a standard representation for functional logic programs where the strategy of pattern matching is explicitly represented by case expressions. Since source programs can be easily translated into the flat form [12], we omit further details about the transformation of source programs into flat programs but define the syntax of flat programs before we present our transformation scheme.

3.1 Syntax of Flat Functional Logic Programs

As a first step we fix the language of polymorphic type expressions. We denote by $\overline{o_n}$ the sequence of objects $o_1, \ldots, o_n$.

[2] http://sebfisch.github.com/explicit-sharing

Definition 1 (Syntax of Type Expressions). *Type expressions are either type variables α or type constructors T applied to type expressions:*

$$\tau ::= \alpha \mid T(\overline{\tau_n})$$

Function types are of the form $\overline{\tau_n} \to \tau$ where $\overline{\tau_n}, \tau$ are type expressions. We denote by $\mathcal{T}$ the set of all function types.

As discussed in Section 1, functional logic programs contain program rules as well as declarations of data types. We summarize type declarations in the notion of a program signature.

Definition 2 (Program signature). *A program signature is a pair (Σ, ty) where $\Sigma = \mathcal{F} \uplus \mathcal{C}$ is the disjoint union of a set $\mathcal{F}$ of function symbols and a set $\mathcal{C}$ of constructor symbols. The mapping $ty : \Sigma \to \mathcal{T}$ maps each symbol in Σ to a function type such that, for all $C \in \mathcal{C}$, $ty(C) = \overline{\tau} \to T(\overline{\alpha})$ for a type constructor T. If $ty(s) = \overline{\tau_n} \to \tau$, then n is called the* arity *of symbol s, denoted by $ar(s)$.*

The signature for the program of Section 1 contains the following symbols

$$\mathcal{C} = \{True, False, Nil, Cons\} \qquad \mathcal{F} = \{append, dup, oneBool, main\}$$

as well as the following mapping of types:

$$\begin{aligned} ty(Nil) &= \to List\ a \\ ty(Cons) &= a, List\ a \to List\ a \\ &\vdots \\ ty(append) &= List\ a, List\ a \to List\ a \\ ty(dup) &= List\ a \to List\ a \\ &\vdots \end{aligned}$$

Next we fix the syntax of programs w.r.t. a given program signature. We consider flat programs where pattern matching is represented by case expressions.

Definition 3 (Syntax of Programs). *Let (Σ, ty) be a program signature specifying the types of all constructor and functions symbols occurring in a program and $\mathcal{X}$ be a set of variables disjoint from the symbols occurring in Σ. A* pattern *is a constructor $C \in \mathcal{C}$ applied to pairwise different variables $\overline{x_n}$ where $n = ar(C)$:*

$$p ::= C(\overline{x_n})$$

Expressions *over (Σ, ty) are variables, constructor or function applications, case expressions, or non-deterministic choices:*

$$\begin{array}{lll} e ::= x & & x \in \mathcal{X} \text{ is a variable} \\ \quad \mid\ C(\overline{e_n}) & & C \in \mathcal{C} \text{ is an } n\text{-ary constructor symbol} \\ \quad \mid\ f(\overline{e_n}) & & f \in \mathcal{F} \text{ is an } n\text{-ary function symbol} \\ \quad \mid\ case\ e\ of\ \{\overline{p_n \to e_n}\} & & p_i \text{ have pairwise different constructors} \\ \quad \mid\ e_1\ ?\ e_2 & & \end{array}$$

Programs *over* (Σ, ty) *contain for each* n*-ary function symbol* $f \in \mathcal{F}$ *one rule of the form* $f(\overline{x_n}) \to e$ *where* $\overline{x_n}$ *are pairwise different variables and* e *is an expression.*

The rules corresponding to the functions `append` and `oneBool` of Section 1 are:

$$\begin{aligned} append(xs, ys) &\to \text{case } xs \text{ of } \{\ Nil \to ys, \\ &\qquad\qquad Cons(z, zs) \to Cons(z, append(zs, ys))\ \} \\ oneBool &\to Cons(True\,?\,False, Nil) \end{aligned}$$

For simplicity, we assume that expressions and programs are well typed w.r.t. the standard Hindley/Milner type system. Furthermore, we assume that there is no shadowing of pattern variables, i.e., the variables occurring in the patterns of a case expression are fresh in the scope of the case expression.

Note that all constructor and function symbols are fully applied. The extension to higher-order functions is discussed separately in Section 4.

3.2 Transforming Data Types

In the following transformations, we assume that `m` is a new type variable that does not occur in the program to be transformed. This type variable will denote the monad that implements non-deterministic evaluations in the target program. Since evaluations can be non-deterministic in all levels of functional logic programs, we have to insert `m` as a new argument in all data types. Thus, we start the definition of our transformation by stating how type expressions of functional logic programs are mapped to Haskell type expressions, adding `m` to all argument types.

Definition 4 (Transforming Types). *The transformation* $tr(\tau)$ *on type expressions* τ *is defined as follows:*

$$\begin{aligned} tr(\overline{\tau_n} \to \tau) &= \texttt{m}\ tr(\tau_1) \to \ldots \to \texttt{m}\ tr(\tau_n) \to \texttt{m}\ tr(\tau) \\ tr(\alpha) &= \alpha \\ tr(T(\overline{\tau_n})) &= \texttt{(}T\ \texttt{m}\ tr(\tau_1) \ldots tr(\tau_n)\texttt{)} \end{aligned}$$

The transformation of data type declarations adds `m` to all constructors:

Definition 5 (Transforming Data Declarations). *For each type constructor* T *of arity* n*, let* $\{\overline{C_k}\} = \{\ C \in \mathcal{C} \mid ty(C) = \cdots \to T(\overline{\alpha_n})\ \}$ *be the set of constructor symbols for this type constructor. Then we transform the definition of type constructor* T *into the following Haskell data type declaration:*

$$\begin{aligned} &\texttt{data}\ T\ \texttt{(m ::} * \to *\texttt{)}\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ C_1\ \texttt{(m}\ tr(\tau_{11})\texttt{)}\ \ldots\ \texttt{(m}\ tr(\tau_{1n_1})\texttt{)} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{|}\ \ldots \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{|}\ C_k\ \texttt{(m}\ tr(\tau_{k1})\texttt{)}\ \ldots\ \texttt{(m}\ tr(\tau_{kn_k})\texttt{)} \end{aligned}$$

where $ty(C_j) = \overline{\tau_{jn_j}} \to T(\overline{\alpha_n})$.

The kind annotation `(m :: * → *)` in the previous definition is necessary for data types which have 0-ary data constructors only (i.e., enumeration types). Without this annotation, a wrong kind for `m` would be deduced in this case due to default assumptions in the Haskell type inferencer. Hence, for data types with at least one non-constant data constructor, the kind annotation can be omitted. For instance, the data types presented in the example of Section 1 are transformed into the following Haskell data type declarations:

```
data Bool (m :: * → *) = True | False
data List m a = Nil | Cons (m a) (m (List m a))
```

3.3 Transforming Functions

As discussed in Section 2, variables that have multiple occurrences in the body of a program rule have to be shared in order to conform to the intended call-time choice semantics of functional logic programs. In order to introduce sharing for such variables in our transformation, we need the notion of the number of free occurrences of a variable in an expression:

Definition 6 (Free Occurrences of a Variable). *The number of free occurrences of variable x in expression e, denoted by $occ_x(e)$, is defined as:*

$$
\begin{aligned}
occ_x(y) &= \begin{cases} 1, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases} \\
occ_x(s(\overline{e_n})) &= \textstyle\sum_{i=1}^{n} occ_x(e_i) \\
occ_x(e_1 \,?\, e_2) &= max\{occ_x(e_1), occ_x(e_2)\} \\
occ_x(\text{case } e \text{ of } \{\overline{p_n \to e_n}\}) &= \begin{cases} 0, & \text{if } x \text{ occurs in some } p_i \ (1 \le i \le n) \\ occ_x(e) + max\{occ_x(e_i) \mid 1 \le i \le n\}, & \text{otherwise} \end{cases}
\end{aligned}
$$

By $vars_n(e)$ we denote the set of variables occurring at least n times in e:

$$vars_n(e) = \{x \in \mathcal{X} \mid occ_x(e) \ge n\}$$

Note that we count multiple occurrences for each possible computation path. Thus, the variable occurrences in the two branches of a non-deterministic choice expression are not added but only the maximum is considered, i.e., if a variable occurs only once in each alternative of a choice, it is not necessary to share it. The same is true for the branches of a case expression.

In order to translate functional logic expressions into Haskell, we have to apply two basic transformations: (1) Introduce sharing for all variables with multiple occurrences (defined by the transformation *sh* below) and (2) Translate non-deterministic into monadic computations (defined by the transformation *tr* below). Note that these transformations are mutually recursive.

Definition 7 (Transforming Expressions). *The transformation $sh(e)$ introduces sharing for all variables with multiple occurrences in the expression e:*

$$sh(e) = share(vars_2(e), tr(e))$$

$$share(\{\overline{x_k}\}, e) = \begin{cases} \texttt{share}\ x_1\ \texttt{>>=}\lambda x_1\ \to \\ \vdots \\ \texttt{share}\ x_k\ \texttt{>>=}\lambda x_k\ \to \\ e \end{cases}$$

For the sake of simplicity, we do not rename the variables when introducing sharing but exploit the scoping of Haskell, i.e., the argument x_i of `share` *is different from the argument x_i in the corresponding lambda abstraction.*

Transformation tr replaces non-deterministic choices by monadic operations and introduces sharing for the pattern variables of case expressions, if necessary:

$$\begin{aligned} tr(x) &= x \\ tr(f(\overline{e_n})) &= \texttt{(}f\ tr(e_1)\ \ldots\ tr(e_n)\texttt{)} \\ tr(C(\overline{e_n})) &= \texttt{(return (}C\ tr(e_1)\ \ldots\ tr(e_n)\texttt{))} \\ tr(e_1\,?\,e_2) &= \texttt{(mplus}\ tr(e_1)\ tr(e_2)\texttt{)} \\ tr(case\ e\ of\ \{\overline{p_n \to e_n}\}) &= \begin{cases} \texttt{(}tr(e)\ \texttt{>>=}\lambda\texttt{x}\ \to \\ \texttt{case x of} \\ \quad p_1\ \to sh(e_1) \\ \quad \vdots \qquad\qquad\qquad\qquad where\ \texttt{x}\ fresh \\ \quad p_n\ \to sh(e_n) \\ \quad \texttt{\_}\ \to \texttt{mzero)} \end{cases} \end{aligned}$$

Note that patterns of case expressions p_i must also be translated into their curried form in Haskell, i.e., each pattern $p_i = C(\overline{x_k})$ is translated into $C\ x_1 \ldots x_k$, but we omit this detail in the definition of tr for the sake of simplicity.

Now we are ready to describe the transformation of program rules by transforming the rule's right-hand side. In addition, we have to add the necessary class dependencies in the type of the defined function as discussed in Section 2.

Definition 8 (Transforming Program Rules). *Let (Σ, ty) be a program signature and $f(\overline{x_n}) \to e$ a rule of a functional logic program. We transform this rule into the following Haskell definition, where $\alpha_1, \ldots, \alpha_k$ are all type variables occurring in $ty(f)$:*

f `::` `(MonadPlus m, Sharing m,`
`Shareable m` α_1`, ..., Shareable m` α_k`)` $\Rightarrow tr(ty(f))$
$f\ x_1\ \ldots\ x_n$ `=` $sh(e)$

According to the transformation scheme, the rules corresponding to operations `append`, `dup`, and `oneBool` (cf. Section 1) are translated to the Haskell definitions:

```
append :: (MonadPlus m, Sharing m, Shareable m a) ⇒
          m (List m a) → m (List m a) → m (List m a)
append xs ys = xs >>= λ l →
  case l of Nil        → ys
            Cons z zs  → return (Cons z (append zs ys))
dup :: (MonadPlus m, Sharing m, Shareable m a) ⇒
       m (List m a) → m (List m a)
dup xs = share xs >>= λ xs → append xs xs
oneBool :: (MonadPlus m, Sharing m) ⇒ m (List m (Bool m))
oneBool = return (Cons (mplus (return True) (return False))
                       (return Nil))
```

4 Extensions

Up to now, we have described a basic transformation of a first-order kernel language. In this section, we discuss extensions of this transformation scheme.

4.1 Higher-Order Programs

Higher-order programs can be translated with an extension of our transformation scheme. We omit some details like the transformation of higher-order function and data types due to lack of space.

In functional (logic) languages, functions are first class citizens which means that functions can have other functions both as argument and as result. In order to add higher-order features to our source language, we extend it with lambda abstractions and higher-order applications:

$$e ::= \cdots \mid \lambda x \to e \mid apply(e_1, e_2)$$

We still require applications of function and constructor symbols to respect the arity of the corresponding symbol. Over-applications can be expressed using *apply* and partial applications can be transformed into applications of lambda abstractions. For example, the partial application "`append oneBool`" in Curry would be expressed as

$$apply(\lambda xs \to \lambda ys \to append(xs, ys), oneBool)$$

in our source language. Note that we do not use the simpler representation

$$\lambda ys \to append(oneBool, ys)$$

which has a different semantics in Curry because *oneBool* would not be shared if this lambda abstraction is duplicated.

We use the function `iterate` as an example for a higher-order function:

```
iterate :: (a → a) → a → List a
iterate f x = Cons x (iterate f (f x))
```

The function `iterate` yields a list of iterated applications of a given function to a value. In the definition, both arguments of `iterate` are shared. Therefore, the transformation scheme of Section 3.3 would introduce sharing as follows:

```
iterate :: (MonadPlus m, Sharing m, Shareable m a) ⇒
           m (m a → m a) → m a → m (List m a)
iterate f x = share f >>= λ f →
              share x >>= λ x →
              return (Cons x (iterate f (apply f x)))
```

The `apply` function is used to transform the higher-order application of the variable "`f`" to "`x`" and is implemented as follows:

```
apply :: MonadPlus m ⇒ m (m a → m b) → m a → m b
apply f x = f >>= λf → f x
```

In order to translate the Curry expression "`iterate (append oneBool) Nil`", we transform the partial application of append as illustrated above and then apply the following rule to transform lambda abstractions:

$tr(\lambda x \to e) =$ `return (λ x →share x >>= λx →` $tr(e)$`)`

Note, that we precautionary share every argument of a lambda abstraction regardless of whether it is shared in the body or not. This is necessary because the lambda abstraction itself could be duplicated and the argument must be shared also if only duplicated indirectly along with the lambda abstraction. We cannot share already supplied arguments when partial applications are duplicated because we reuse Haskell's higher-order features and, hence, partial Curry applications are represented as Haskell functions that we cannot inspect.

Here is the transformed version of the above call to `iterate`:

```
iterate (apply (return (λ xs → share xs >>= λ xs →
                 return (λ ys → share ys >>= λ ys →
                 append xs ys)))
               oneBool)
        (return Nil)
```

This translation shares the result of `oneBool` just like the original Curry expression when the argument "`f`" of `iterate` is duplicated. Therefore, the result of this call is an infinite list of boolean lists of increasing length where all elements are either `True` or `False` but no list contains both `True` and `False`.

4.2 An Optimized Transformation Scheme

In this section we present a technique to optimize the programs obtained from the transformation in Section 3.3. The basic idea is that the original transformation scheme may introduce sharing too early. To keep things simple, Definitions 7 and 8 introduce sharing at the beginning of a rule or the case branches, respectively. This scheme is straightforward and a similar scheme is used in PAKCS [11]. When implementing the transformation presented here, we observed that sharing could also be introduced individually for each variable as "late" as possible. Consequently, the ideas presented in this section could also be employed to improve existing compilers like that of PAKCS.

What does "late sharing" mean? Reconsider the transformed `iterate` function given in Section 4.1. Due to the nature of `iterate`, the result is a potentially infinite list. Therefore, in any terminating program the context of a call to iterate will only demand the `x` of the result but not the value of the expression `iterate f (f x)`. It is clear that for yielding `x` in this case there is no need to share `f` (again). Thus, sharing `f` later will improve the resulting code:

```
iterate f x = share x >>= λ x →
              return (Cons x (share f >>= λ f →
                              iterate f (apply f x)))
```

The example also shows that the optimization requires to introduce sharing individually for each variable.

How can we obtain this optimization in general? The idea is that the transformation of expressions needs some information about which variables occur in its context. Whenever the situation arises that for a term $s(\overline{e_n})$ a variable occurs in more than one of the e_n but not in the context, we have to introduce sharing for x right around the result of transforming $s(\overline{e_n})$. Therefore, the transformation tr is extended by an additional argument indicating the set of variables occurring in the context. These ideas are formalized in the following definitions.

First we formalize the idea that variables "occur in more than one" of a sequence of given expressions.

Definition 9. $multocc(\overline{e_n}) = \{x \mid \exists i \neq j : x \in vars_1(e_i) \cap vars_1(e_j)\}$

The optimizing transformation scheme for expressions is then expressed in the following definition. There, the transformation gets as an additional argument the set of variables for which sharing was already introduced. For a variable that does not occur in that set, sharing will be introduced in two situations: (a) before an application if it occurs in more than one argument, or (b) before a case expression "case e *of* $\{\overline{p_n \to e_n}\}$" if it occurs in e *and in at least one* of the branches $\overline{e_n}$.

Definition 10 (Optimized Transformation of Expressions). *The optimized transformation of an expression e w.r.t. a set of variables V, denoted $tr(V, e)$, is defined as follows (the transformation share is as in Definition 7):*

$$
\begin{aligned}
tr(V, x) &= x \\
tr(V, s(\overline{e_n})) &= share(S, s'(tr(V \cup S, e_1), \ldots, tr(V \cup S, e_n))) \\
\quad where \qquad S &= multocc(\overline{e_n}) \setminus V \\
s'(\overline{t_n}) &= \begin{cases} \texttt{(}s\ t_1\ \ldots\ t_n\texttt{)} & , \ if \ \ s \in \mathcal{F} \\ \texttt{(return (}s\ t_1\ \ldots\ t_n\texttt{))} & , \ if \ \ s \in \mathcal{C} \end{cases} \\
tr(V, e_1 \,?\, e_2) &= \texttt{(mplus } tr(V, e_1)\ tr(V, e_2)\texttt{)} \\
tr(V, case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}) &= share(S, \left\{ \begin{array}{l} \texttt{(}tr(V \cup S, e)\ \texttt{>>=}\lambda \texttt{x} \rightarrow \\ \texttt{case x of} \\ \quad p_1 \quad \rightarrow tr(V \cup S, e_1) \\ \quad \ldots \\ \quad p_n \quad \rightarrow tr(V \cup S, e_n) \\ \quad \texttt{\_} \quad \rightarrow \texttt{mzero)} \end{array} \right\}) \\
\quad where \qquad & \texttt{x}\ fresh \\
& S = (\textstyle\bigcup_{i=1}^{n} multocc(e, e_i)) \setminus V
\end{aligned}
$$

According to the idea that the additional argument of the transformation represents the set of variables for which sharing was already introduced, the initial value of the argument should be the empty set as expressed in the next definition.

Definition 11 (Optimized Transformation of Functions). *The optimized transformation of a function defined by a rule $f(\overline{x_n}) \rightarrow e$ is similar to Definition 8 but uses the transformation from Definition 10.*

$f\ \ x_1\ \ \ldots\ \ x_n\ \texttt{=}\ tr(\emptyset, e)$

5 Conclusions and Related Work

In this paper we presented a scheme to translate functional logic programs into pure Haskell programs. The difficulty of such a translation is the fact that non-deterministic results can occur in any level of a computation, i.e., arbitrarily deep inside data structures. This problem is solved by transforming all computations into monadic ones, i.e., all argument and result values of functions and data constructors have monadic types w.r.t. a "non-determinism monad", i.e. a `MonadPlus` instance. Furthermore, the monad must support explicit sharing in order to implement the sharing of potentially non-deterministic arguments, which is necessary for a non-strict functional logic language with call-time choice. As a result, we obtain target programs which are parametric w.r.t. the concrete evaluation monad, i.e., one can execute the same target code with different search strategies, choose between call-time choice or run-time choice for parameter passing, or add additional run-time information to implement specific tools.

Considering related work, many schemes to compile lazy functional logic programs into various target languages have been introduced. Due to the nature of these languages, former approaches can be categorized with respect to the target

language: (a) schemes targeting a logic programming language (b) compiling to a lazy functional language (c) generating code for a especially devised abstract machine (implemented in an imperative language, typically). Considering (a) there have been several attempts to target Prolog and make use of the logic features of that host language, e.g., the $\mathcal{TOY}$ system [16], and PAKCS [11]. With respect to the implementation presented here, a system based on Prolog can not easily support different search strategies simply because Prolog does not support them. On the other hand, Prolog implementations normally offer various constraint solvers, which can therefore be easily integrated in a functional logic system. Typically, however, these integrations suffer from the fact that constraint solvers for Prolog are implemented with respect to a strict semantics. The resulting issues with a lazy semantics make such an integration not as seamless as possible. With respect to (b) there have been various proposals to implement logic programming in a functional language. As discussed in detail in [8], most of these proposals do not adequately represent laziness. The exception to this is KiCS [7], which employs a different translation scheme to compile Curry to Haskell. In contrast to the scheme presented here, the current implementation of KiCS employs side effects for the implementation of logic features. Consequently, the resulting programs can not be optimized by standard Haskell compilers. In addition, the attempt to introduce a parallel search strategy to KiCS has failed due to the side effects. In contrast to our approach, however, KiCS provides sharing of deterministic expressions across non-deterministic computations [7]. Regarding (c), sharing across non-determinism is also provided by FLVM, the abstract machine described in [5], which is implemented in Java. The FLVM has undergone substantial changes from the implementation described in [5], and can still be considered to be in an experimental state. Finally, the MCC [18] is based on an abstract machine implemented in C. The MCC provides a programatic approach to support different search strategies, i.e., the Curry programmer can influence the search strategy by calling primitive operators provided in this system.

Bundles [15] improve laziness in purely functional non-deterministic computations similar to our translation of data types. The type for bundles is a transformed list data type restricted to the list monad without non-deterministic list elements. Nesting non-determinism inside constructors plays an essential role in achieving full abstraction in a semantics for constructor systems under run-time choice [17].

By representing non-determinism explicitly using monads, we can collect results of non-deterministic computations in a deterministic data structure which is called encapsulated search [6,3]. The monadic formulation of lazy non-determinism provides a new perspective on the problems described in previous work on encapsulated search and possibilities for future work.

In a next step, we will implement the transformation scheme into a complete compiler for Curry in order to test it on a number of benchmarks. Although it is clear that one has to pay a price (in terms of execution efficiency) for the high-level parametric target code, initial benchmarks, presented in [8], demonstrate that

the clean target code supports optimizations of the Haskell compiler so that the monadic functional code can compete with other more low level implementations. Based on such an implementation, it would be interesting to test it with various monad instances in order to try different search strategies, in particular, parallel strategies, or to implement support for run-time tools, like observation tools, debuggers etc. Furthermore, one could also use the monad laws of [8] together with our transformation scheme in order to obtain a verified implementation of Curry.

References

1. S. Antoy. Constructor-based conditional narrowing. In *Proc. of PPDP 2001*, pages 199–206. ACM Press, 2001.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
3. S. Antoy and B. Braßel. Computing with subspaces. In M. Leuschel and A. Podelski, editors, *Proc. of PPDP 2007*, pages 121–30. ACM, 2007.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. of the 22nd Int. Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
5. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th Int. Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
6. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
7. B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.
8. S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy non-deterministic programming. In *Proc. of ICFP 2009*, pages 11–22. ACM, 2009.
9. J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
10. M. Hanus. Multi-paradigm declarative languages. In *Proc. of the Int. Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
11. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2008.
12. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
13. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
14. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
15. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Bundles pack tighter than lists. In *Draft Proc. of Trends in Functional Programming 2007*, pages XXIV–1–XXIV–16, 2007.

16. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
17. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A fully abstract semantics for constructor systems. In *RTA '09: Proc. of the 20th Int. Conference on Rewriting Techniques and Applications*, pages 320–334. Springer, 2009.
18. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer, 1999.
19. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
20. P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.

Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL

George Giorgidze, Henrik Nilsson

Functional Programming Laboratory
University of Nottingham
United Kingdom
{ggg,nhn}@cs.nott.ac.uk

Abstract. This paper explores how to implement an *iteratively staged* domain-specific language (DSL) by embedding into a functional language. The domain is modelling and simulation of physical systems where models are expressed in terms of non-causal differential-algebraic equations; i.e., sets of constraints solved through numerical simulation. What sets our language apart is that the equational constraints are *first class entities* allowing for an evolving model structure characterised by *repeated generation* of updated constraints. Hence iteratively staged. Our DSL can thus be seen as a combined functional and constraint programming language, albeit a two-level one, with the functional language chiefly serving as a meta language. However, the two levels do interact throughout the simulation. The embedding strategy we pursue is a mixture of deep and shallow, with the deep embedding enabling just-in-time (JIT) compilation of the constraints as they are generated for efficiency, while the shallow embedding is used for the remainder for maximum leverage of the host language. The paper is organised around a specific DSL, but our implementation strategy should be applicable for iteratively staged languages in general. Our DSL itself is further a novel variation of a declarative constraint programming language.

1 Introduction

Embedding is a powerful and popular way to implement domain-specific languages (DSLs) [1]. Compared with implementing a language from scratch, extending a suitable general-purpose programming language, the *host language*, with notions and vocabulary addressing a particular application or problem domain tends to save a lot of design and implementation effort.

There are two basic approaches to language embeddings: *shallow* and *deep*. In a shallow embedding, domain-specific notions are expressed directly in host-language terms, typically through a higher-order combinator library. This is a light-weight approach that makes it easy to leverage the facilities of the host language. However, the syntactic freedom is limited, and the static semantics of the embedded language must be relatively close to that of the host language for an embedding to be successful. In contrast, a deep embedding is centred around

a *representation* of embedded language terms that then are given meaning by interpretation or compilation. This is a more heavy-weight approach, but also more flexible. In particular, for optimisation or compilation, it is often necessary to inspect terms, suggesting a deep embedding. The two approaches can be combined to draw on the advantages of each. This leads to *mixed-level* embedding.

In this paper, we explore how to embed a language, Hydra [2], for non-causal modelling and simulation of physical systems into a functional programming language. In this application domain, systems are modelled by constraints expressed as undirected Differential Algebraic Equations (DAEs). These equations are solved by specialised combined symbolic and numerical simulation methods. A defining aspect of Hydra is that the equations are *first-class entities* in a functional language layer, providing very flexible means for expressing model composition and evolving model structure. Specifically, in response to *events*, which occur at discrete points in time, the simulation is stopped and, *depending* on results thus far, (partly) new equations are *generated* describing a (partly) new problem to be solved. We refer to this kind of DSL as *iteratively staged* to emphasise that the *domain* is characterised by repeated program generation and execution. Iterative staging makes it possible to model classes of systems in Hydra that current main-stream non-causal modelling and simulation languages cannot handle [3]. Section 2 exemplifies one such system.

Hydra can be seen as a functional and constraint or logical programming language in that it combines a functional and relational approach to programming. However, the integration of the two approaches is less profound than in, say, functional logic languages based on residuation or narrowing [4]. Hydra is a two-level language, where the functional part to a large extent serves as a meta language. However, the two layers do interact throughout the simulation.

We have chosen Haskell as the host language, or, more precisely, Haskell with Glasgow Haskell Compiler (GHC) extensions, GHC's quasiquoting facility [5,6] being one reason for this choice. Because performance is a primary concern in the domain, the simulation code corresponding to the current equations has to be compiled. As this code is determined *dynamically*, this necessitates *just-in-time* (JIT) compilation. We use a deep embedding for this part of the language along with the Low-Level Virtual Machine (LLVM)[1]: a language-independent, portable, optimising, compiler back-end with JIT support. In contrast, we retain a shallow embedding for the parts of the embedded language concerned with high-level, symbolic computations to get maximum leverage from the host language. Note that we are not concerned with (hard) *real-time* performance here: we are prepared to pay the price of brief "pauses" for symbolic processing and compilation in the interest of minimising the computational cost of the actual simulation that typically dominates the overall costs by a wide margin.

An alternative might have been to use a *multi-staged* host language like MetaOCaml [7]. The built-in run-time code generation capabilities of the host language would then have been used instead of relying on an external code generation framework. We have so far not explored this approach as we wanted to

[1] `http://llvm.org/`

have tight control over the generated code. Also, not predicating our approach on a multi-staged host language means that some of our ideas and implementation techniques can be more readily deployed in other contexts, for example to enhance the capabilities of existing implementations of non-causal languages.

Compilation of Embedded DSLs (EDSLs) is today a standard tool in the DSL-implementer's tool chest. The seminal example is the work by Elliott et al. on compiling embedded languages, specifically the image synthesis and manipulation language Pan [8]. Pan, like our language, provides for program generation by leveraging the host language combined with compilation to speed up the resulting performance-critical computations. However, the program to be compiled is generated once and for all, meaning the host language acts as a powerful but fundamentally conventional macro language: program generation, compilation, and execution is a process with a fixed number of stages.

As Hydra is iteratively staged, the problems we are facing are in many ways different. Also, rather than acting merely as a powerful meta language that is out of the picture once the generated program is ready for execution, the host language is in our case part of the dynamic semantics of the embedded language through the shallow parts of the embedding. With this paper, we thus add further tools to the DSL tool chest for embedding a class of languages that hitherto has not been studied much. Specifically, our contributions are:

- a case study of mixed-level embedding of iteratively staged DSLs;
- using JIT compilation to implement an iteratively staged EDSL efficiently.

Additionally, we consider static type checking in the context of iterative staging and quasiquoting-based embedding. While Hydra is specialised, we believe the ideas underlying the implementation are of general interest, and that Hydra itself should be of interest to programming language researchers interested in languages that combine functional and relational approaches to programming. The implementation is available on-line[2] under the open source BSD license.

The rest of the paper is organised as follows. In Section 2, we introduce non-causal modelling and our language Hydra in more detail. Section 3 explains the Haskell embedding of Hydra and Section 4 then describes how iteratively staged programs are executed. Related work is discussed in Section 5. Finally, Section 6 gives conclusions.

2 Background

This section provides an introduction to Functional Hybrid Modelling (FHM) [2] and to our specific instance Hydra [9,3]. We focus on aspects that are particularly pertinent to the present setting. The reader is referred to the earlier papers on FHM and Hydra for a more general treatment.

[2] `http://cs.nott.ac.uk/~ggg/`

2.1 Functional Hybrid Modelling

Functional Hybrid Modelling (FHM) [2] is a new approach to designing *non-causal modelling languages* [10] supporting *hybrid systems*: systems that exhibit both continuous and discrete dynamic semantics. This class of languages is intended for modelling and simulation of systems that can be described by *Differential Algebraic Equations* (DAEs). Examples include electrical, mechanical, hydraulic, and other physical systems, as well as their combinations. *Non-causal*[3] in this context refers to treating the equations as being *undirected*: an equation can be used to solve any of the variables occurring in it. This is in contrast to *causal* modelling languages where equations are restricted to be *directed*: only "known" variables on one side of the equal sign, and only "unknown" variables on the other. Note that the domain of the variables are *time-varying values* or *signals*: functions of continuous time.

The advantages of non-causal languages over causal ones include that models are more *reusable* (the equations can be used in many ways) and more *declarative* (the modeller can focus on *what* to model, worrying less about *how* to model it to enable simulation) [10]. These are crucial advantages in many modelling domains. As a result, a number of successful non-causal modelling languages have been developed. Modelica[4] is a prominent, state-of-the-art example.

However, one well-known weakness of current non-causal languages is that their support for modelling *structurally dynamic systems*, systems where the equations that describe the dynamic behaviour change at discrete points in time, usually is limited. There are a number of reasons for this. A fundamental one is that languages like Modelica, to facilitate efficient simulation, are designed on the assumption that the model is translated into simulation code once and for all, *before* simulation starts.

The idea of FHM is to enrich a purely functional language with a few key abstractions for supporting hybrid, non-causal modelling. In particular, first-class *signal relations*, relations on signals described by undirected DAEs, provide support for non-causal modelling, and *dynamic switching* among signal relations that are *computed* at the point when they are being "switched in" provides support for describing highly structurally dynamic systems [2].

Our hypothesis is that the FHM approach will result in non-causal modelling languages that are more expressive than the current ones, yet have relatively simple, declarative semantics. Results so far have been promising. The capability to compute and use new signal relations during simulation has already allowed us to non-causally model and simulate some systems that e.g. Modelica cannot handle [3]. We present one such example in the following. The dynamic computation of and switching among signal relations is, of course, also what makes FHM iteratively staged.

[3] Do not confuse this with *temporal* causality. A system is temporally causal if its output only depends on present and past input, and temporally non-causal if the output depends on future input.

[4] `http://www.modelica.org/`

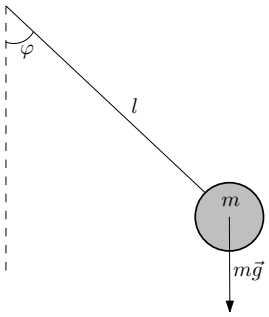

Fig. 1. A pendulum subject to gravity.

2.2 Hydra by Example: The Breaking Pendulum

To introduce Hydra, let us model a physical system whose structural configuration changes abruptly during simulation: a simple pendulum that can break at a specified point in time; see Figure 1. The pendulum is modelled as a point mass m at the end of a rigid, mass-less rod, subject to gravity $m\vec{g}$. If the rod breaks, the mass will fall freely. This makes the differences between the two configurations sufficiently large that e.g. Modelica does not support non-causal modelling of this system. Instead, if simulation across the breaking point is desired, the modeller is forced to model the system in a causal, less declarative way.

There are two levels to Hydra: the *functional level* and the *signal level*. The functional level is concerned with the definition of ordinary functions operating on time-invariant values. The signal level is concerned with the definition of relations between signals, the *signal relations*, and, *indirectly*, the definition of the *signals* themselves as solutions satisfying these relations.

Signal relations are *first-class* entities at the functional level. The type of a signal relation is parametrised on a *descriptor* of the types of the signals it relates: essentially a tuple of the types carried by the signals. For example, the type of a signal relation relating three real-valued signals is $SR\ (Real, Real, Real)$.

Signals, in contrast to signal relations, are *not* first-class entities at the functional level. However, crucially, *instantaneous values* of signals can be propagated back to the functional level, allowing the future system structure to depend on signal values at discrete points in time.

The definitions at the signal level may freely refer to entities defined at the functional level as the latter are time-invariant, known parameters as far as solving the equations are concerned. However, the opposite is not allowed: time-varying entities are confined to the signal level. The only signal-level notion that exists at the functional level is the *time-invariant* signal relation.

Hydra is currently implemented as an embedding in Haskell using *quasiquoting* [5,6]. This means Haskell provides the functional level almost for free through shallow embedding. In contrast, the signal level is realised through deep embedding: signal relations expressed in terms of Hydra-specific syntax are, through the

```
type Coordinate = (Double, Double)
type Velocity = (Double, Double)
type Body = (Coordinate, Velocity)

g :: Double
g = 9.81

freeFall :: Body → SR Body
freeFall ((x0, y0), (vx0, vy0)) = [$hydra|
  sigrel ((x, y), (vx, vy)) where
    init (x, y)         = ($x0$, $y0$)
    init (vx, vy)       = ($vx0$, $vy0$)
    (der x, der y)      = (vx, vy)
    (der vx, der vy)    = (0, −$g$)
|]
```

```
pendulum :: Double → Double
              → SR Body
pendulum l phi0 = [$hydra|
  sigrel ((x, y), (vx, vy)) where
    init phi      = $ phi0 $
    init der phi  = 0
    init vx       = 0
    init vy       = 0
    x             =    $ l $ ∗ sin phi
    y             = − $ l $ ∗ cos phi
    (vx, vy)      = (der x, der y)
    der (der phi)
      + ($g / l$) ∗ sin phi = 0
|]
```

Fig. 2. Models of the two modes of the pendulum.

quasiquoting machinery, turned into an internal representation that then is compiled into simulation code. This, along with the reasons for using quasiquoting, is discussed in more detail in an earlier paper [9]. However, that paper only treated *structurally static* systems.

Figure 2 shows how to model the two modes of the pendulum in Hydra. The type *Body* denotes the position and velocity of an object, where position and velocity both are 2-dimensional vectors represented by pairs of doubles. Each model is represented by a function that maps the *parameters* of the model to a relation on signals; i.e., an instance of the defining system of DAEs for specific values of the parameters. In the unbroken mode, the parameters are the length of the rod *l* and the initial angle of deviation *phi0*. In the broken mode, the signal relation is parametrised on the initial state of the body.

[$*hydra*| and |] are the open and close quasiquotes. Between them, we have signal-level definitions expressed in our custom syntax. The keyword **sigrel** starts the definition of a signal relation. It is followed by a pattern that introduces *signal variables* giving local names to the signals that are going to be constrained by the signal relation. This pattern thus specifies the *interface* of a signal relation.

Note the two kinds of variables: the functional level ones representing *time-invariant* parameters, and the signal-level ones, representing *time-varying* entities, the signals. Functional-level fragments, such as variable references, are spliced into the signal level by enclosing them between antiquotes, $. On the other hand time-varying entities are not allowed to escape to the functional level (meaning signal-variables are not in scope between antiquotes).

After the keyword **where** follow the equations that define the relation. These equations may introduce additional signal variables as needed. Equations marked by the keyword **init** are initialisation equations used to specify initial conditions. The operator *der* indicates differentiation with respect to time of the signal-valued expression to which it is applied.

```
pendulumBE :: Double → Double → Double
              → SR (Body, E Body)
pendulumBE t l phi0 = [$hydra|
  sigrel (((x, y), (vx, vy)), event e) where
    $ pendulum l phi0 $ ⋄ ((x, y), (vx, vy))
    event e = ((x, y), (vx, vy))
          when time = $t $
|]
```

(a) Pendulum extended with a breaking event

```
breakingPendulum :: SR Body
breakingPendulum =
  switch
    (pendulumBE 10 1 (pi / 4))
    freeFall
```

(b) Composition using switch

Fig. 3. The breaking pendulum

The non-causal nature of Hydra can be seen particularly clearly in the last equation of the unbroken mode that simply states a constraint on the angle of deviation and its second derivative, without making any assumption regarding which of the two time-varying entities is going to be used to solve for the other (both g and l are time-invariant functional-level variables).

To model a pendulum that breaks at some point, we need to create a composite model where the model that describes the dynamic behaviour of the unbroken pendulum is replaced, at the point of breaking, by the model describing a free falling body. These two submodels must be suitably joined to ensure the continuity of both the position and velocity of the body of the pendulum.

To this end, the *switch*-combinator, which forms signal relations by temporal composition, is used:

$$switch :: SR\ (a, E\ b) \rightarrow (b \rightarrow SR\ a) \rightarrow SR\ a$$

The composite behaviour is governed by the first signal relation until an *event* of type b occurs ($E\ b$ in the type signature above). At this point, the second argument to switch is applied to the value carried by the event to *compute* the signal relation that is going to govern the composite behaviour from then on. Event signals are *discrete-time* signals, signals that are only defined at (countably many) discrete points in time, as opposed to the continuous-time signals that (conceptually) are defined everywhere on a continuous interval of time. Each point of definition of an event signal is known as an *event occurrence*. Unlike continuous-time signals, the causality of event signals is always fixed.

Figure 3 shows how *switch* is used to construct a model of a breaking pendulum. The *pendulum* model is first extended into a signal relation *pendulumBE* that also provides the event signal that defines when the pendulum is to break: see figure 3(a). In our case, an event is simply generated at an *a priori* specified point in time, but the condition could be an arbitrary time-varying entity. The value of the event signal is the state (position and velocity) of the pendulum at that point, allowing the succeeding model to be initialised so as to ensure the continuity of the position and velocity as discussed above.

To bring the equations of *pendulum* into the definition of *pendulumBE*, *pendulum* is first applied to the length of the pendulum and the initial angle of deviation at the *functional level* (within antiquotes), thus computing a signal relation. This *relation* is then *applied*, at the signal level, using the *signal relation application operator* $\diamond$. This instantiates the equations of *pendulum* in the context of *pendulumBE*. Unfolding signal relation application in Hydra is straightforward: the actual arguments (signal-valued expressions) to the right of the signal relation application operator $\diamond$ are simply substituted for the corresponding formal arguments (signal variables) in the body of the signal relation to the left of $\diamond$. See [9] for further details.

Finally, a model of the breaking pendulum can be composed by switching form *pendulumBE* to *freeFall*: see figure 3(b). Note that the switching event carries the state of the pendulum at the breaking point as a value of type *Body*. This value is passed to *freeFall*, resulting in a model of the pendulum body in free fall initialised so as to ensure the continuity of its position and velocity.

In our particular example, the pendulum is only going to break once. In other words, there is not much iteration going on, and it would in principle (with a suitable language design) be straightforward to generate code for both modes of operation prior to simulation. However, this is not the case in general. For example, given a parametrised signal relation:

$$sr1 :: Double \rightarrow SR\ ((Double, Double), E\ Double)$$

we can recursively define a signal relation *sr* that describes an overall behaviour by "stringing together" the behaviours described by *sr1*:

$$sr :: Double \rightarrow SR\ (Double, Double)$$
$$sr\ x = switch\ (sr1\ x)\ sr$$

In this case, because the number of instantiations of *sr1* in general cannot be determined statically (and because each instantiation can depend on the parameter in arbitrarily complex ways), there is no way to generate all code prior to simulation. However, the pendulum example is simple and suffice for illustrative purposes. Moreover, despite its simplicity, it is already an example with which present non-causal languages struggle, as mentioned above.

In practical terms, the *switch*-combinator is a somewhat primitive way of describing variable model structure. Our aim is to enrich Hydra with higher-level constructs as descried in the original FHM paper [2]. The basic aspects of the implementation should, however, not change much.

3 Embedding

In this section, we describe the Haskell embedding of Hydra in further detail. First, we introduce a Haskell data type that represents an embedded signal relation. This representation is untyped. We then introduce typed combinators that ensures that only well-typed signal relations can be constructed.

The following data type is the central, untyped representation of signal relations. There are two ways to form a signal relation: either from a set of defining equations, or by composing signal relations temporally:

```
data SigRel =
    SigRel       Pattern [Equation]
  | SigRelSwitch SigRel  (Expr → SigRel)
```

The constructor *SigRel* forms a signal relation from equations. Such a relation is represented by a pattern and the list of defining equations. The pattern serves the dual purpose of describing the *interface* of the signal relation in terms of the types of values carried by the signals it relates and their time domains (continuous time or discrete time/events), and of introducing names for these signals for use in the equations. Patterns are just nested tuples of signal variable names along with indications of which ones are event signals: we omit the details. The list of equations constitute a system of Differential Algebraic Equations (DAEs)[5] that defines the signal relation by expressing constraints on the (signal) variables introduced by the pattern and any additional local variables.

The *switch*-combinator forms a signal relation by temporal composition of two signal relations. Internally, such a temporal composition is represented by a signal relation constructed by *SigRelSwitch*. The first argument is the signal relation that is initially active. The second argument is the function that, in the case of an event occurrence from the initially active signal relation, is used to compute a new signal relation from the value of that occurrence. This new signal relation then becomes the active one, replacing the initial signal relation.

Note the use of a mixture of shallow and deep techniques of embedding. The embedded function in a signal relation constructed by *SigRelSwitch* corresponds to the shallow part of the embedding. The rest of the data types constitute a deep embedding, providing an explicit representation of language terms for further symbolic processing and ultimately compilation, as we will see in more detail below. The following data type represents equations. There are four different kinds:

```
data Equation =
  EquationInit    Expr   Expr        | EquationEq          Expr   Expr |
  EquationEvent String Expr Expr | EquationSigRelApp SigRel Expr
```

Initialisation equations, constructed by *EquationInit*, provide initial conditions. They are thus only in force when a signal relation instance first becomes active.

Equations constructed by *EquationEq* are basic equations imposing the constraint that the valuations of the two expressions have to be equal for as long as the containing signal relation instance is active (e.g., equations like $der\ (der\ x) = 0$). Equations constructed by *EquationEvent* define event signals; i.e., they represent equations like **event** $e = (x, y)$ **when** $time = 3$. These equations are directed.

[5] Although not necessarily a *fixed* such system as these equations may refer to signal relations that contain switches.

The string is the name of the defined event signal. The first expression gives the value of the event signal at event occurrences. The second expression defines these occurrences. An event occurs whenever the signal represented by this expression *crosses* 0. For the above example, the expression defining the event occurrences would thus be $time - 3$.

The fourth kind of equation is signal relation application, *EquationSigRelApp*, i.e. equations like $sr \diamond (x, y + 2)$. This brings all equations of a signal relation into scope by instantiating them for the expressions to which the relation is applied.

Finally, the representation of expressions is a standard first-order term representation making it easy to manipulate expressions symbolically (e.g. computing symbolic derivatives) and compiling expressions to simulation code:

```
data Expr = ExprUnit | ExprReal Double  | ExprVar String | ExprTime |
            ExprTuple Expr Expr [Expr] | ExprApp Function [Expr]
data Function = FuncDer | FuncNeg | FuncAdd | FuncnSub | FuncMul | ...
```

We use quasiquoting, a recent Haskell extension implemented in Glasgow Haskell Compiler (GHC), to provide a convenient surface syntax for signal relations. We have implemented a quasiquoter that takes a string in the concrete syntax of Hydra and generates Haskell code that builds the signal relation in the mixed-level representation described above. GHC executes the quasiquoter for each string between the quasiquotes before type checking.

While the internal representation of a signal relation is untyped, Hydra itself is typed, and we thus have to make sure that only type-correct Hydra programs are accepted. As Hydra fragments are generated dynamically, during simulation, we cannot postpone the type checking to after program generation. Nor can we do it early, at quasiquoting time, at least not completely, as no type information from the context around quasiquoted program fragments are available (e.g., types of antiquoted Haskell expressions). In the current version of Hydra, only domain specific scoping rules (e.g., all constrained signal variables must be declared) are checked at the stage of quasiquoting. Fortunately, the type system of the present version of Hydra is fairly simple; in particular, Hydra is simply typed, so by using the standard technique of phantom types, the part of the type checking that requires type information outside the quasiquotes is delegated to the host language type checker [11].

A phantom type is a type whose type constructor has a parameter that is not used in its definition. We define phantom type wrappers for the untyped representation as follows:

```
data SR a       = SR       SigRel
data PatternT a = PatternT Pattern
data ExprT a    = ExprT    Expr
data E a
```

Phantom types can be used to restrict a function to building only type-correct domain-specific terms. For example, a typed combinator *sigrel* can be defined in the following way:

$$sigrel :: PatternT\ a \rightarrow [Equation] \rightarrow SR\ a$$
$$sigrel\ (PatternT\ p)\ eqs = SR\ (SigRel\ p\ eqs)$$

As can be seen, the type of the pattern that defines the interface of the signal relation is what determines its type.

Similarly, we define a typed combinator *switch*:

$$switch :: SR\ (a, E\ b) \rightarrow (b \rightarrow SR\ a) \rightarrow SR\ a$$

E is a type constructor with no constituent data constructors. It is used to type patterns that introduce event signals. The data for the event signals are constructed using event equations.

A signal relation that is defined using the *switch* combinator is structurally dynamic. However, the type of the *switch* combinator statically guarantees that its type (i.e., its interface) remains unchanged. Thus, a structurally dynamic signal relation can be used in a signal relation application just like any other signal relation.

Well-typed equations are constructed using combinators in a similar way:

$$equationEq :: ExprT\ a \rightarrow ExprT\ a \rightarrow Equation$$
$$equationSigRelApp :: SR\ a \rightarrow ExprT\ a \rightarrow Equation$$

Typed combinators for the remaining parts of the language, including *Pattern* and *Expr*, are defined using the same technique.

Under the hood the representation is still untyped. However, if only the typed combinators are exposed for building of signal relations, it is guaranteed that only well-typed terms can be constructed. The quasiquoter of Hydra has only access to typed combinators for building signal relations.

Symbolic transformations (e.g., symbolic differentiation and flattening) on embedded language terms work with the untyped representation. These transformations need to be programmed with care as the Haskell type checker cannot verify that the transformations are type preserving.

Several type system extensions of Haskell (e.g., generalised algebraic data types, existential types, and type families) make alternative techniques for typing EDSLs possible. One alternative would be to directly construct signal relations in typed representation and implement the symbolic transformations on the typed representation. While this approach requires more work from the EDSL implementer, it provides additional correctness guarantees (e.g., the Haskell type checker could be used to verify that transformations are type preserving). We have not yet evaluated suitability of Haskell type system for such undertaking and opted for simpler, untyped representation.

4 Simulation

In this section we describe how an iteratively staged Hydra program is run. The process is illustrated in Figure 4 and is conceptually divided into four stages. In

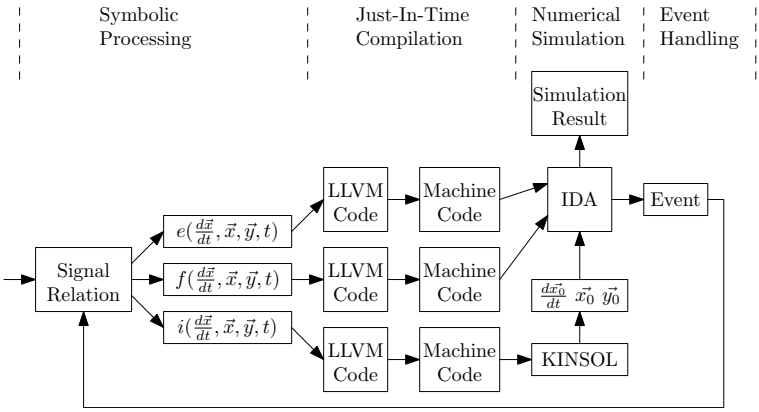

Fig. 4. Execution model of Hydra

the first stage, a signal relation is flattened and subsequently transformed into a mathematical representation suitable for numerical simulation. In the second stage, this representation is JIT compiled into efficient machine code. In the third stage, the compiled code is passed to a numerical solver that simulates the system until the end of simulation or an event occurrence. In the fourth stage, in the case of an event occurrence, the event is analysed, a corresponding new signal relation is computed and the process is repeated from the first stage. In the following, each stage is described in more detail.

As a first step, all signal variables are renamed to give them distinct names. This helps avoiding name clashes during *flattening*, signal relation application unfolding, and thus simplifies this process. Having carried out this preparatory renaming step, all signal relation applications are unfolded until the signal relation is completely flattened.

Further symbolic processing is then performed to transform the flattened signal relation into a form that is suitable for numerical simulation. In particular, derivatives of compound signal expressions are computed symbolically. In the case of higher-order derivatives, extra variables and equations are introduced to ensure that all derivatives in the flattened system are first order.

Finally, the following equations are generated at the end of the stage of symbolic processing: $i(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0, t = t_0$ (1), $f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0$ (2), and $e(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0$ (3). Here, $\vec{x}$ is a vector of differential variables, $\vec{y}$ is a vector of algebraic variables, t is time, and t_0 is the starting time for the current set of equations. Equation 1 determines the initial conditions for Equation 2 (i.e., the values of $\frac{d\vec{x}}{dt}$,$\vec{x}$ and $\vec{y}$ at time t_0). Equation 2 is the main DAE of the system that needs to be integrated in time starting from the initial conditions. Equation 3 specifies the event conditions (signals crossing 0).

As the functions i, f, and e are invoked from within inner loops of the solver, they have to be compiled into machine code for efficiency: any interpretive over-

head here would be considered intolerable by practitioners for most applications. However, as Hydra allows the equations to be changed in arbitrary ways *during* simulation, the equations have to be compiled whenever they change, as opposed to only prior to simulation. Our Hydra implementation employs JIT machine code generation using the compiler infrastructure provided by LLVM. The functions i, f and e are compiled into LLVM instructions that in turn are compiled by the LLVM JIT compiler into native machine code. Function pointers to the generated machine code are then passed to the numerical solver.

The numerical suite used in the current implementation of Hydra is called SUNDIALS[6]. The components we use are KINSOL, a nonlinear algebraic equation systems solver, and IDA, a differential algebraic equation systems solver. The code for the function i is passed to KINSOL that numerically solves the system and returns initial values (at time t_0) of $\frac{d\vec{x}}{dt}$,$\vec{x}$ and $\vec{y}$. These vectors together with the code for the functions f and e are passed to IDA that proceeds to solve the DAE by numerical integration. This continues until either the simulation is complete or until one of the events defined by the function e occurs. Event detection facilities are provided by IDA.

At the moment of an event occurrence (one of the signals monitored by e crossing 0), the numerical simulator terminates and presents the following information to an event handler: Name of the event variable for which an event occurrence has been detected, time t_e of the event occurrence and instantaneous values of the signal variables (i.e., values of $\frac{d\vec{x}}{dt}$, $\vec{x}$ and $\vec{y}$ at time t_e).

The event handler traverses the original unflattened signal relation and finds the event value expression (a signal-level expression) that is associated with the named event variable. In the case of the breaking pendulum model, the expression is $((x, y), (vx, vy))$. This expression is evaluated by substituting the instantaneous values of the corresponding signals for the variables. The event handler applies the second argument of the *switch* combinator (i.e., the function to compute the new signal relation to switch into) to the functional-level event value. In the case of the breaking pendulum model, the function *freeFall* is applied to the instantaneous value of $((x, y), (vx, vy))$. The result of this application is a new signal relation. The part of the original unflattened signal relation is updated by replacing the old signal relation with the new one. The flat system of equations for the previous mode and the machine code that was generated for it by the LLVM JIT compiler are discarded. The simulation process for the updated model continues from the first stage and onwards.

In previous work [3], we conducted benchmarks to evaluate the performance of the proposed execution model. The initial results are encouraging. For a small system with handful of equations (e.g., the breaking pendulum) the total time spent on run-time symbolic processing and code generation is only a couple of hundredth of a second. To get an initial assessment of how well our approach scales, we also conducted a few large scale benchmarks (thousands of equations). These demonstrated that the overall performance of the execution model seems to scale well. In particular, time spent on run-time symbolic processing and JIT

[6] `http://www.llnl.gov/casc/sundials/`

compilation increased roughly linearly in the number of equations for these tests. The results also demonstrate that the time spent on JIT compilation dominates over the time spent on run-time symbolic processing. Above all, our benchmarks indicated that the time for symbolical processing and compilation remained modest in absolute terms, and thus should be relatively insignificant compared with the time for simulation in typical applications.

In the current implementation of Hydra, a new flat system of equations is generated at each mode switch without reusing the equations of the previous mode. It may be useful to identify exactly what has changed at each mode switch, thus enabling the reuse of *unchanged* equations and associated code from the previous mode. In particular, this could reduce the burden placed on the JIT compiler, which in our benchmarks accounted for most of the switching overheads. Using such techniques, it may even be feasible to consider our kind of approach for structurally dynamic (soft) *real-time* applications.

Our approach offers new functionality in that it allows non-causal modelling and simulation of structurally dynamic systems that simply cannot be handled by static approaches. Thus, when evaluating the feasibility of our approach, one should weigh the overheads against the limitation and inconvenience of not being able to model and simulate such systems non-causally.

5 Related Work

The deep embedding techniques used in the Hydra implementation for domain-specific optimisations and efficient code generation draws from the extensive work on compiling staged domain-specific embedded languages. Examples include Elliott et al. [8] and Mainland et al. [6]. However, these works are concerned with compiling programs all at once, meaning the host language is used only for meta-programming, not for running the actual programs.

The use of quasiquoting in the implementation of Hydra was inspired by Flask, a domain-specific embedded language for programming sensor networks [6]. However, we had to use a different approach to type checking. A Flask program is type checked by a domain-specific type checker *after* being generated, just before the subsequent compilation into the code to be deployed on the sensor network nodes. This happens at *host language run-time*. Because Hydra is iteratively staged, we cannot use this approach: we need to move type checking back to *host language compile-time*. The Hydra implementation thus translates embedded programs into typed combinators at the stage of quasiquoting, charging the host language type checker with checking the embedded terms. This ensures only well-typed programs are generated at run-time.

Lee et al. are developing a DSL embedded in Haskell for data-parallel array computations on a graphics processing unit (GPU) [12]. GPU programs are first-class entities. The embedded language is being designed for run-time code generation, compilation and execution, with results being fed back for use in further host language computations. Thus, this is another example of what we

term iterative staging. At the time of writing, the implementation is interpreted. However, a JIT compiler for a GPU architecture is currently being developed.

The FHM design was originally inspired by Functional Reactive Programming (FRP) [13], particularly Yampa [14]. A key difference is that FRP provides *functions* on signals whereas FHM generalises this to *relations* on signals. FRP can thus be seen as a framework for *causal* simulation, while FHM supports non-causal simulation. Signal functions are first class entities in most incarnations of FRP, and new ones can be computed and integrated into a running system dynamically. This means that these FRP versions, including Yampa, also are examples of iteratively staged languages. However, as all FRP versions supporting highly dynamic system structure so far have been interpreted, the program generation aspect is much less pronounced than what is the case for FHM.

In the area of non-causal modelling, Broman's work on the Modelling Kernel Language (MKL) has a number of similarities to FHM [15]. MKL provides a λ-abstraction for defining functions and an abstraction similar to **sigrel** for defining non-causal models. Both functions and non-causal models are first-class entities in MKL, enabling higher-order, non-causal modelling like in FHM. However, support for structural dynamism has not yet been considered.

Non-causal languages that do support more general forms of structural dynamism than the current mainstream ones include MOSILAB[7], a Modelica extension, and Sol [16], a Modelica-like language. MOSILAB has a compiled implementation, but insists all structural configurations are predetermined to enable compilation once and for all, prior to simulation. Sol is less restrictive, but currently only has an interpreted implementation. Both MOSILAB and Sol could thus benefit from the implementation techniques described in this paper.

6 Conclusions

In this paper we presented a novel implementation approach for non-causal modelling and simulation languages supporting structural dynamism. Our approach was to embed an iteratively staged DSL in Haskell, using a mixed-level embedding to capitalise maximally on the host language while simultaneously enabling an efficient implementation through JIT compilation. The iterative staging allows us to model systems that current, main-stream, non-causal languages cannot handle without resorting to causal modelling. As far as we are aware, this is the first compiled implementation of a non-causal modelling language that supports highly structurally dynamic systems, demonstrating the practical feasibility of such a language as compilation of simulation code is considered essential for most practical applications for reasons of performance.

The use of the EDSL approach was instrumental to achieve the above. By reusing the features of the host language and its tool chain, we could focus our efforts on the problems that are specific to non-causal modelling and simulation. We also note that LLVM has worked really well for our purposes. Compilation

[7] `http://www.mosilab.de/`

of iteratively staged embedded languages does not seem to have attracted much attention thus far. We hope the implementation techniques we have developed will be useful to others who are faced with implementing such languages.

Acknowledgements. This work was supported by EPSRC grant EP/D064554/1. We would like to thank Neil Sculthorpe and the anonymous reviewers for their thorough and constructive feedback that helped to improve the paper.

References

1. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse. (1998)
2. Nilsson, H., Peterson, J., Hudak, P.: Functional hybrid modeling. In: Proceedings of 5th International Workshop on Practical Aspects of Declarative Languages. (2003)
3. Giorgidze, G., Nilsson, H.: Higher-order non-causal modelling and simulation of structurally dynamic systems. In: Proceedings of the 7th International Modelica Conference. (2009)
4. Hanus, M., Kuchen, H., Moreno-Navarro, J.J.: Curry: A truly functional logic language. In: Proceedings Workshop on Visions for the Future of Logic Programming. (1995)
5. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. (2007)
6. Mainland, G., Morrisett, G., Welsh, M.: Flask: Staged functional programming for sensor networks. In: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming. (2008)
7. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. (2004)
8. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. In: Semantics, Applications, and Implementation of Program Generation. (2000)
9. Giorgidze, G., Nilsson, H.: Embedding a functional hybrid modelling language in Haskell. In: Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages. (2008)
10. Cellier, F.E.: Object-oriented modelling: Means for dealing with system complexity. In: Proceedings of the 15th Benelux Meeting on Systems and Control. (1996)
11. Rhiger, M.: A foundation for embedded languages. ACM Trans. Program. Lang. Syst. (2003)
12. Lee, S., Chakravarty, M., Grover, V., Keller, G.: GPU kernels as data-parallel array computations in Haskell. In: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods. (2009)
13. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of International Conference on Functional Programming. (1997)
14. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop. (2002)
15. Broman, D., Fritzson, P.: Higher-order acausal models. In: Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. (2008)
16. Zimmer, D.: Introducing Sol: A general methodology for equation-based modeling of variable-structure systems. In: Proceedings of the 6th International Modelica Conference. (2008)

An Access Control Language based on Term Rewriting and Description Logic*

Michele Baggi[1], Demis Ballis[2], and Moreno Falaschi[1]

[1] Dip. di Scienze Matematiche e Informatiche
Pian dei Mantellini 44, 53100 Siena, Italy.
{baggi,moreno.falaschi}@unisi.it
[2] Dip. Matematica e Informatica
Via delle Scienze 206, 33100 Udine, Italy.
demis@dimi.uniud.it

Abstract. This paper presents a rule-based, domain specific language for modeling access control policies which is particularly suitable for managing security in the semantic web, since (i) it allows one to evaluate authorization requests according to semantic information retrieved from remote knowledge bases; (ii) it supports semantic-based policy composition, delegation and closure via flexible operators which can be defined by security administrators in a pure declarative way with little effort. The operational engine of the language smoothly integrates description logic into standard term rewriting giving support to reasoning capabilities which are particularly useful in this context, since they allow one to naturally combine and reuse data extracted from multiple knowledge bases. Such a rewrite engine can be used to evaluate authorization requests w.r.t. a policy specification as well as to formally check properties regarding the security domain to be protected. The language we propose has been implemented in a prototypical system, which is written in Haskell. Some case studies have been analyzed to highlight the potentiality of our approach.

1 Introduction

The widespread use of web-based applications provides an easy way to share and exchange data as well as resources over the Internet. In this context, controlling the user's ability to exercise access privileges on distributed information is a crucial issue, which requires adequate security and privacy support. In recent years, there has been a considerable attention to distributed access control, which has rapidly led to the development of several domain specific languages for the specification of access control policies in such heterogeneous environments: among those, it is worth mentioning the standard XML frameworks XACML [20] and WS-Policy [24]. Nevertheless, the proposed approaches offer a limited support to the emerging semantic web technologies which are very often embedded into

* This work has been partially supported by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004.

modern web applications. More generally, we can affirm that the security impact of such semantic-aware applications have been not sufficiently investigated to date.

In the semantic web, resources are annotated with machine-understandable metadata which can be exploited by intelligent agents to infer semantic information regarding the resources under examination. Therefore, in this context, application's security aspects should depend on the semantic nature of the entities into play (e.g. resources, subjects). In particular, it would be desirable to be able to specify access control requirements about resources and subjects in terms of the rich metadata describing them.

In this paper, we present a rule-based, domain specific language well-suited to manage security of semantic web applications. As a matter of fact, it allows security administrators to tightly couple access control rules with knowledge bases (modeled using Description Logic (DL) [1]) that provide semantic-aware descriptions of subjects and resources.

The operational mechanism of our language is based on a rewriting-like mechanism integrating DL into term rewriting [2]. Specifically, the standard rewrite relation is equipped with reasoning capabilities which allow us to extract semantic information from (possibly remote) knowledge bases in order to evaluate authorization requests. In this setting, access control policies are modeled as sets of rewrite rules, called *policy rules*, which may contain queries expressed in an appropriate DL language. Hence, evaluating an authorization request —specifying the intention of a subject to gain access to a given resource— boils down to rewriting the initial request using the policy rules until a decision is reached (e.g. *permit*, *deny*, *notApplicable*).

Since policy composition is an essential aspect of access control in collaborative and distribute environments ([6,12,18,7]), our language is also endowed with policy assembly facilities which allow us to glue together several simpler access control policies into a more complex one. To this respect, our language is expressive enough to model all the XACML[20] composition algorithms as well as other conflict-resolution, closure and delegation criteria.

Finally, it is worth noting that our formal framework is particularly suitable for the analysis of policy's domain properties such as *cardinality constraints* and *separation of duty*. As our rewriting mechanism combines term rewriting with description logic, the analysis of policy specifications can fruitfully exploit both rewriting techniques and DL reasoning capabilities.

Related work. Term rewriting has been proven successful in formalizing access control of systems. For instance, [4] demonstrates that term rewriting is an adequate formalism to model Access Control Lists as well as Role-based Access Control (RBAC) policies. Moreover, it shows how properties of the rewrite relation can enforce policy correctness properties. Also issues regarding policy composition have been investigated within the term rewriting setting. For example, [6] formalizes a higher-order rewrite theory in which access control policies are combined together by means of higher-order operators; then, modularity properties of the theory are used to derive the correctness of the global policy.

An alternative methodology for policy composition is presented in [12]: in this approach, composition is achieved by using rewriting strategies that combine rewrite rules specifying individual policies in a consistent, global policy specification. Unfortunately, all the presented rewriting-like approaches —albeit very useful in several contexts— do not offer any semantic web support. In particular, the specification and analysis of access control policies cannot take advantage of any semantic information (e.g. RDFs, OWL descriptions, *etc.*).

In recent years, some efforts have been made towards the integration of semantic-aware data into access control languages. For instance, [8] presents an extension of XACML supporting semantic metadata modeled as RDF statements. In [13,10,15] security ontologies are employed to allow parties to share a common vocabulary for exchanging security-related information. In particular, [15] describes a decentralized framework which allows one to reuse and combine distinct policy languages by means of semantic web technologies. As opposed to our approach, [15] does not define an access control language for policy specification, rather it supports the integration and management of existing policy languages. PeerTrust [14] provides a sophisticated mechanism for gaining access to secure information on the web by using semantic annotations, policies and automated trust negotiation. Basically, trust is established incrementally by gradually disclosing credentials and requests for credentials. For the time being our framework does not include a protocol for trust negotiation, however we would like to implement an iterative disclosure approach *à la* PeerTrust in the future.

Description logic (specifically, $\mathcal{ALQ}$ logic) has been used in [26] to represent and reason about the RBAC model: basically, this approach encodes the RBAC model into a knowledge base expressed by means of DL axioms, then DL formulae are checked within the knowledge base to verify policy properties (e.g. separation of duty). Ontologies modeled by means of DL statements have been used in [18] in order to specify and check a very expressive subset of the XACML language. In this approach, (a part of) XACML is first mapped to a suitable description logic, then a DL reasoner is employed for analysis tasks such as policy comparison, verification and querying. DL ontologies have also been used in [23] and [16] to describe policy languages for the specification of access restrictions and obligations.

To the best of our knowledge, our work is the first attempt to develop an integrated framework combining term rewriting and semantic assertions (modeled as DL knowledge bases) for access control purposes. We do believe that such an integration allows one to fully take advantage of rewriting and DL techniques to ease specification, composition, and analysis of access control policies.

2 Preliminaries

By $\mathcal{V}$ we denote a countably infinite set of variables and Σ denotes a *signature*, that is, a set of *operators* each of which has a fixed associated arity. In this paper, we assume that variables and symbols are typed following the standard many-sorted discipline [21]. As usual, given a set of sorts $\mathcal{S}$, and $\tau_1, \ldots, \tau_n, \tau \in \mathcal{S}$, a variable $x \in \mathcal{V}$ of type τ is denoted by $x :: \tau$, while by $f :: \tau_1 \ldots \tau_n \to \tau$ we

represent the type of the operator $f \in \Sigma$ of arity n. $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the *non-ground term algebra* and the *term algebra* built on $\Sigma \cup \mathcal{V}$ and Σ, respectively.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. The empty sequence Λ denotes the root position. Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t that are rooted by symbols in S. $t_{|u}$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm rooted at the position u replaced by r. A term t is *ground*, if no variable occurs in t. A term t is *linear*, if no variable appears more than once in t. Let $\Sigma \cup \{\circ\}$ be a signature such that $\circ \notin \Sigma$. The symbol $\circ$ is called a *hole*. A *context* is a term $\gamma \in \tau(\Sigma \cup \{\circ\}, \mathcal{V})$ with zero or more holes $\circ$. We write $\gamma[\,]_u$ to denote that there is a hole at position u of γ. By notation $\gamma[\,]$, we define an arbitrary context (where the number and the positions of the holes are clarified *in situ*), while we write $\gamma[t_1, \ldots t_n]$ to denote the term obtained by filling the holes appearing in $\gamma[\,]$ with terms $t_1, \ldots, t_n$. Syntactic equality is represented by $\equiv$.

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots\}$ is a sort-preserving mapping from the set of variables $\mathcal{V}$ into the set of terms $\tau(\Sigma, \mathcal{V})$ satisfying the following conditions: (*i*) $x_i \neq x_j$, whenever $i \neq j$, (*ii*) $x_i\sigma = t_i$, $i = 1, \ldots, n$ and (*iii*) $x\sigma = x$, for all $x \in \mathcal{V} \setminus \{x_1, \ldots, x_n\}$. By ε we denote the *empty* substitution. An *instance* of a term t is defined as $t\sigma$, where σ is a substitution. By $Var(s)$ we denote the set of variables occurring in the syntactic object s.

Description logic. Description Logics (DLs) are decidable logic formalisms for representing knowledge of application domains and reasoning about it. In DL, domains of interest are modeled as knowledge bases (i.e. ontologies) by means of concepts (classes), individuals (instances of classes) and roles (binary predicates).

In this section, we present (a subset of) the decidable description logic $\mathcal{SHOIQD}_n^-$ underlying the OWL-DL [25] framework. Due to lack of space, here we only describe the syntax of those DL constructors which are relevant to this work. An explanation of the semantics of the considered DL constructs and the main DL reasoning services (e.g. satisfiability and subsumption) can be found in the technical report available at [3]. For a full discussion about OWL and DL formalisms, please respectively refer to [9] and [1].

Let Σ_D be a signature containing all the symbols of the considered DL language. A concept C is defined using the following constructors of Σ_D.

$$C ::= A \mid \neg C_1 \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C_1 \mid \forall R.C_1 \mid \geqslant_n R \mid \leqslant_n S \mid \{a\}$$

where A represents an atomic concept, C_1 and C_2 are concepts, R is a role, n is a natural number, and a is an individual. We use $\bot$ (resp., $\top$) to abbreviate the concept $C \sqcap \neg C$ (resp., $C \sqcup \neg C$). Besides, we make use of the role constructor $(\cdot)^-$ to define the *inverse* of a role R. The inverse of a role R is still a role and is denoted by R^-. Concepts are related to each other using *terminological axioms* of the form $C_1 \sqsubseteq C_2$. A *TBox* is a finite set of terminological axioms. Given a concept C, a role R, and two individuals a and b, a *concept assertion* is an expression of the form $C(a)$, while the expression $R(a, b)$ denotes a *role assertion*. An *ABox* is a finite set of concept and role assertions. A *knowledge base* $\mathcal{K}$ is a pair $(\mathrm{TB}, \mathrm{AB})$, where TB is a TBox and AB is an ABox.

We consider the basic reasoning services $satisfiable(C)$ and $subsume(C_1, C_2)$, which can be executed against a knowledge base by a DL reasoner. The former checks concept satisfiability for a given concept C, while the latter checks whether concept C_1 subsumes concept C_2. We also consider the additional reasoning services $instances(C)$ and $instance(a, C)$[3]. The former retrieves all individuals belonging to the concept C from a knowledge base, while the latter checks whether the individual a belongs to the concept C. A *DL query* is an expression $DL(\mathcal{K}, r)$ where $\mathcal{K}$ is a knowledge base and r is a reasoning service. Basically, a DL query, when evaluated, executes a given reasoning service against a knowledge base and returns a result, which can be either a boolean constant or a list of values. We call *boolean* (respectively, *non-boolean*) DL query any DL query whose executions return a boolean value (respectively, a list of values).

Example 1. Let $\mathcal{H}$ be a knowledge base modeling a healthcare domain. Assume that $\mathcal{H}$ includes the atomic concepts: *patient*, *physician*, *guardian*, *admin*; and the role *assignedTo*, which establishes who are the people designated to take care of a given patient. Moreover, the ABox of $\mathcal{H}$ is populated by the following concept and role assertions:

$patient(charlie), patient(laura), physician(alice), physician(bob), guardian(frank),$
$admin(john), assignedTo(alice, charlie), assignedTo(frank, charlie).$

Now, consider the following DL queries Q_1, Q_2 and Q_3:

$DL(\mathcal{H}, instance(bob, physician \sqcap \exists assignedTo.\{charlie\}))$,
$DL(\mathcal{H}, subsumes(\neg physician, admin))$, $DL(\mathcal{H}, instances(guardian \sqcup physician)$

Q_1 and Q_2 are boolean DL queries, while Q_3 is a non-boolean DL query. More specifically, Q_1 asks $\mathcal{H}$ whether *bob* is the designated physician of patient *charlie*, in this case the execution of Q_1 returns *false*, since *charlie*'s designated physician is *alice*. Q_2 checks whether concept *admin* is subsumed by concept $\neg$*physician*, that amounts to saying there are no administrators who are also physicians. Finally, the evaluation of Q_3 computes the list [*frank*, *alice*, *bob*] representing all the individuals belonging to the union of concepts *guardian* and *physician*.

Description logics are ground formalisms, that is, logic formulae do not contain variables. In particular, variables cannot appear in DL queries. As we will see in the following sections, sometimes it might be convenient to generalize the notion of DL query by admitting the use of variables. In this way, DL queries may be (i) easily reused, and (ii) ground values associated with the considered variables may be computed at run-time. In light of these considerations, we define the notion of DL query template as follows. A *reasoning service template* is defined as a reasoning service that may contain variables playing the role of placeholders for concepts, roles, and individuals. A *DL query template* is an expression $DL(\mathcal{K}, r)$ where $\mathcal{K}$ is a knowledge base, and r is a reasoning service template. It is worth noting that a DL query template cannot be executed by a DL reasoner, since

[3] The Pellet DL reasoner [22], which we used in our experiments, supports all the mentioned reasoning and additional services.

only ground formulae can be evaluated by the reasoner. Therefore, in order to make a DL query template executable, it is necessary to make it ground by instantiating all its variables with ground values. Later on, we will show that such an instantiation process is inherently supported by our rewriting-like mechanism (see Section 3.1).

Example 2. Consider the knowledge base $\mathcal{H}$ of Example 1. The following expression is a DL query template: $DL(\mathcal{H}, instances(physician \sqcap \exists assignedTo.\{X\}))$ where X is a variable representing a generic individual. Note that the evaluation of the template depends on the concrete individual assigned to X. For instance, if X was bound to the individual *charlie*, the result of the evaluation would be $\{alice\}$, while we would obtain the empty set in the case when X was associated with the individual *laura*.

3 Policy Specification Language

Let Σ_D be the signature defining all the symbols of the Description Logic language of Section 2, that is, DL operators, constants, reasoning service constructs, *etc.* A *policy signature* Σ_P is a signature such that $\Sigma_D \subseteq \Sigma_P$ and Σ_P is equipped with the following types: *Subject*, *Action*, *Object*, and *Decision*. A term $t \in \tau(\Sigma_P)$ is *pure* if no DL query appears in t.

Our specification language considers a very general access control model, in which policies authorize or prohibit *subjects* to perform *actions* over *objects*. We formalize subjects, actions and objects as terms of a given term algebra which is built out of a policy signature. More formally, given a policy signature Σ_P, a *subject* (resp. *action*, *object*, *decision*) is any pure term in $\tau(\Sigma_P)$ whose type is *Subject* (resp. *Action*, *Object*, and *Decision*).

The policy behavior is specified by means of rules which are basically rewrite rules whose right-hand sides may contain DL query templates used to extract information from the knowledge bases of interest. Roughly speaking, a policy rule allows one to define what is permitted and what is forbidden using a rewriting-like formalism. Moreover, policy rules can also encode conflict-resolution as well as rule composition operators which can implicitly enforce a given policy behavior (see Section 4).

Definition 1. *Let Σ_P be a policy signature. A* policy rule *is a rule of the form $\lambda \mapsto \gamma[q_1, \ldots, q_n]$ where $\lambda \in \tau(\Sigma_P, \mathcal{V})$, $\gamma[\,]$ is a context in $\tau(\Sigma_P \cup \{\circ\}, \mathcal{V})$, and each $q_i \in \tau(\Sigma_D, \mathcal{V})$, $i = 0, \ldots, n$ is a DL query template such that $Var(\gamma[q_1, \ldots, q_n]) \subseteq Var(\lambda)$.*

Given a policy rule $r \equiv f(t_1, \ldots, t_n) \mapsto \gamma[q_1, \ldots, q_n]$, f is called *defined* symbol for r. Policies are specified by means of sets of policy rules which act over subjects, actions and objects as formally stated in Definition 2.

Definition 2. *Let Σ_P be a policy signature. An* access control policy *(or simply* policy*) is a triple $(\Sigma_P, P, auth)$, where (i) P is a set of policy rules;*

(ii) $auth \in \Sigma_P$ is a defined symbol for some policy rule in P such that $auth :: Subject\ Action\ Object \rightarrow Decision$. The symbol $auth$ is called policy evaluator.

In general, decisions are modeled by means of the constants *permit* and *deny* which respectively express accessibility and denial of a given resource. Sometimes, it is also useful to include a constant *notApp* to formalize policies which do not allow to derive an explicit decision (that is, policies which are not applicable). This is particularly convenient when composing policies (see [6,12]). Moreover, thanks to term representation, we can formulate decisions which convey much more information than a simple authorization or prohibition constant. For instance, the starting time and the duration of a given authorization can be easily encoded into a term (e.g. *permit(Starting-time,Duration)*) [12].

$$
\begin{array}{ll}
auth(p(n(X), age(Z)), read, rec(n(Y))) \mapsto & (1)\\
\quad case(DL(\mathcal{H}, instance(X, patient)) \text{ and } Z > 16 \text{ and } X = Y) \Rightarrow permit & \\
\quad case(DL(\mathcal{H}, instance(X, (\exists assignedTo.\{Y\} \sqcap guardian) \sqcup physician)) \Rightarrow permit & \\
\quad case(DL(\mathcal{H}, instance(X, admin)) \Rightarrow deny & \\
auth(p(n(X), age(Z)), write, rec(n(Y))) \mapsto & (2)\\
\quad case(DL(\mathcal{K}, instance(X, admin)) \Rightarrow deny & \\
\quad case(DL(\mathcal{H}, instance(X, physician \sqcap \exists assignedTo.\{Y\}))) \Rightarrow permit & \\
case(true : Condlist, D : Dlist) \mapsto D & (3)\\
case(false : Condlist, D : Dlist) \mapsto case(Condlist, Dlist) & (4)
\end{array}
$$

Fig. 1. An access control policy for medical record protection

The following example, which is inspired by the XACML specification in [20], shows how to model a policy for the protection of medical records.

Example 3. Consider the knowledge base $\mathcal{H}$ of Example 1 and assume that *and*, $=$, $>$ are built-in, infix boolean operators provided with their usual meanings. Let P_H be the set of policy rules of Figure 1. Then, $\mathcal{P}_\mathcal{H} \equiv (\Sigma_H, P_H, auth)$, where Σ_H is a policy signature containing all the symbols occurring in P_H, is an access control policy formalizing the following plain English security constraints:

- A person, identified by her name, may read any medical record for which she is the designated patient provided that she is over 16 years of age.
- A person may read any medical record for which she is the designated guardian or if she is a physician.
- A physician may write any medical record for which she is the designated physician.
- An administrator shall not be permitted to read or write medical records.

For the sake of readability, we added some syntactic sugar to rules (1) and (2) of P_H in Figure 1. Specifically, the function call $case(cond_1 : \ldots : cond_n, decision_1 : \ldots decision_n)$ has been expanded as follows: $case\ cond_1 \Rightarrow decision_1 \ldots\ case\ cond_n \Rightarrow decision_n$. Note that policy rules (3) and (4) of P_H defining the *case* function enforce a conflict resolution operator which simulates the XACML *first-applicable* criterion; that is, only the first condition which is fulfilled derives a decision. Therefore, in this scenario, a 20 year old administrator who is also a patient would be authorized to read her medical record, although administrators are in general not allowed to read any record.

3.1 Policy Evaluation Mechanism

A query *evaluation* function is a mapping *eval* which takes a DL query as input and returns a data term (typically a boolean value or a list of values belonging to the knowledge base of interest). Thus, by $eval(DL(\mathcal{K}, r))$, we denote the *evaluation* of the DL query $DL(\mathcal{K}, r)$, that is, the result of the execution of the reasoning service r against the knowledge base $\mathcal{K}$.

Example 4. Consider the DL queries Q_1, Q_2 and Q_3 of Example 1. Then, $eval(Q_1) = false$, $eval(Q_2) = true$, $eval(Q_3) = [frank, alice, bob]$.

Definition 3. *Let $(\Sigma_P, P, auth)$ be an access control policy and $t, t' \in \tau(\Sigma_P)$ be two pure terms. Then, t* d-rewrites *to t' w.r.t. P (in symbols, $t \mapsto_P t'$) iff there exist a rule $\lambda \mapsto \gamma[q_1, \ldots, q_n] \in P$, a position $u \in O_{\Sigma_P}(t)$, and a substitution σ such that $t_{|u} \equiv \lambda\sigma$ and $t' \equiv t[\gamma\sigma[eval(q_1\sigma), \ldots, eval(q_n\sigma)]]_u$.*

When P is clear from the context, we simply write $\mapsto$ instead of $\mapsto_P$. Transitive ($\mapsto^+$), and transitive and reflexive ($\mapsto^*$) closures of relation $\mapsto$, as well as the notions of termination and confluence of $\mapsto$ are defined in the usual way.

Definition 4. *Let $(\Sigma_P, P, auth)$ be an access control policy. Let s be a subject, a be an action, o be an object, and d be a decision in $\tau(\Sigma_P)$. We say that $(\Sigma_P, P, auth)$* derives *the decision d w.r.t. (s, a, o) iff there exists a finite d-rewrite sequence $auth(s, a, o) \mapsto_P^+ d$.*

Example 5. Consider the access control policy $\mathcal{P}_\mathcal{H}$ of Example 3. Then $\mathcal{P}_\mathcal{H}$ derives the decision *permit* w.r.t. $(p(n(alice), age(35)), write, rec(charlie))$, since $auth(p(n(alice), age(35)), write, rec(charlie)) \mapsto case(false : true, deny : permit) \mapsto^+ permit$

The specification language of Section 3 allows one to formalize arbitrary access control policies which may be ambiguous or not completely defined, since the rewrite relation $\mapsto$ might be non-terminating or non-confluent. To avoid such problems, we assume the access control policies meet the following properties:

Totality. Let $\mathcal{P}$ be a policy. $\mathcal{P}$ is *total* iff $\mathcal{P}$ derives a decision d for any triple (s, a, o), where s is a subject, a is an action, and o is an object (that is, there exists a finite d-rewrite sequence $auth(s, a, o) \mapsto^+ d$, for any (s, a, o)).

Consistency. Let $\mathcal{P}$ be a policy. $\mathcal{P}$ is *consistent* iff $\mathcal{P}$ derives only one decision d for any triple (s, a, o), where s is a subject, a is an action, and o is an object (that is, if $auth(s, a, o) \mapsto^+ d_1$ and $auth(s, a, o) \mapsto^+ d_2$, then $d_1 \equiv d_2$).

It is worth noting that, in rewrite-based access control, it is common practice to require policies to be total and consistent. Typically, such constraints are enforced by imposing termination, confluence and sufficient completeness of the rewrite systems underlying the access control requirements (e.g. [4,6,12]).

4 Policy operators: Composition, Delegation, and Closure

Policy Composition. In distributed environments (e.g. collaborating organizations, large companies made up of several departments, *etc.*) it is crucial to be able to combine policies in order to protect resources from unauthorized access. Besides, policy composition makes it possible the reuse of security components, which are known to be well specified, to build more complex (and still safe) policies.

In our framework, policy assembling is achieved through policy rules that compose access control policies via policy *combiners*. Basically, policy combiners collect all the decisions taken by local policies and then yield a global decision following the conflict-resolution criterion they encode.

Definition 5. *A policy* combiner *is a triple* $(\Sigma_C, C, comb)$*, where* Σ_C *is a policy signature,* C *is a set of policy rules, and* $comb \in \Sigma_C$ *is a defined symbol for some policy rules in* C *such that* $comb :: [Decision] \to Decision$*. The symbol* $comb$ *is called* combination operator*.*

Roughly speaking, combination operators are applied to lists of decisions which derive from the evaluations of local access control policies. The result of such an application is a single decision corresponding to the evaluation of the composition of the considered policies.

This notion of policy combiner is rather powerful, since it allows security administrators to freely define combination criteria according to their needs. Moreover, it is not difficult to see that policy rules can capture the semantics of all the well known combiners of the action control language XACML[20], namely, *permit-overrides*, *deny-overrides*, *first-applicable*, and *only-one-applicable*. To this respect, Example 6 shows how to formalize the *permit-overrides* operator within our setting. For the specification of the other combiners, please refer to [3].

Example 6. The XACML permit-overrides combiner is defined as follows. Let $d_1, \ldots, d_n$ be a list of decisions which corresponds to the evaluation of n access control policies. If there exists d_i, for some $i = 1, \ldots, n$, equals to *permit*, then, regardless of the other decisions, the combiner returns *permit*.

Let PO be the set of policy rules of Figure 2, where *if cond then exp₁ else exp₂* is assumed to be a built-in conditional construct, and symbols [ ], : are the usual list constructors.

$$
\begin{aligned}
po(d : dList) &\mapsto \text{if } d = permit \text{ then } permit \\
&\qquad \text{else } po\_aux(d, dList) \\
po\_aux(x, [\,]) &\mapsto x \\
po\_aux(x, permit : xs) &\mapsto permit \\
po\_aux(x, deny : xs) &\mapsto po\_aux(deny, xs) \\
po\_aux(x, notApp : xs) &\mapsto po\_aux(x, xs)
\end{aligned}
$$

Fig. 2. *Permit-overrides* combiner

Let Σ_{PO} be a policy signature containing all the symbols occurring in PO. Then, $\mathcal{PO} \equiv (\Sigma_{PO}, PO, po)$ is a policy combiner with combing operator po that models the behavior of the XACML permit-overrides combination criterion.

Starting from (atomic) access control policies, we can assemble more complex policies by applying several policy combiners in a hierarchical way. In other words, access control policies play the roles of basic building blocks which are

glued together by means of policy combiners. Composition of policies is formally defined below.

Definition 6. *Let $(\Sigma_C, C, comb)$ be a policy combiner, s be a subject, a be an action, and o be an object. Then a composition of policies for (s, a, o) is a term $comb(t_1, \ldots, t_n)$ where each t_i, $i = 1, \ldots, n$, is either $auth(s, a, o)$, where auth is the policy evaluator of an access control policy $(\Sigma_P, P, auth)$, or a composition of policies for (s, a, o).*

Basically, evaluating a composition of policies c for a triple (s, a, o) amounts to executing the access control policies and the policy combiners involved in the composition for (s, a, o) by means of the d-rewriting mechanism; that is, we d-rewrite c until we reach a decision. It is worth noting that we cannot simply d-rewrite c w.r.t. the union of all the policy rules involved in the composition, since termination and confluence of the d-rewrite relation are not modular properties. This specifically implies that the termination (resp. confluence) of local policies and combiners does not guarantee the termination (resp. confluence) of the global composition. For instance, it could happen that policy rules defined in distinct local policies $\mathcal{P}_1$ and $\mathcal{P}_2$ interfere in the evaluation of the global composition producing a non-terminating or non-confluent behavior, even if $\mathcal{P}_1$ and $\mathcal{P}_2$ are total and consistent policies. To solve this problem, we follow a bottom-up approach which restricts the application of policy rules in the following way: (i) any authorization request $auth(s, a, o)$ referring to a local policy $\mathcal{P}$ is evaluated using only the policy rules in $\mathcal{P}$; (ii) a combination of policies $comb(t_1, \ldots, t_m)$ referring to a policy combiner $\mathcal{C}$ is evaluated using the policy rules in $\mathcal{C}$ only after the evaluation of terms $t_1, \ldots, t_m$. Such restricted evaluation is formalized in Definition 7 using the following auxiliary functions. Let $\mathcal{P} \equiv (\Sigma_P, P, auth)$ be an access control policy, and $\mathcal{C} \equiv (\Sigma_C, C, comb)$ be a policy combiner, then

$$reduce(auth(s, a, o), P) = d \text{ iff } auth(s, a, o) \mapsto_P^+ d$$
$$reduce(comb(d_1, \ldots, d_n), C) = d \text{ iff } comb(d_1, \ldots, d_n) \mapsto_C^+ d.$$

where $d, d_1, \ldots, d_n$ are decisions.

Definition 7. *Let $(\Sigma_C, C, comb)$ be a policy combiner, s be a subject, a be an action, and o be an object. Then, a composition of policies $comb(t_1, \ldots, t_n)$ for (s, a, o) derives the decision d by evaluating the following function*

$$compute(comb(t_1, \ldots, t_n), C) = reduce(comb(d_1, \ldots, d_n), C)$$
$$\text{where}$$
$$d_i = \begin{cases} reduce(t_i, P) & \text{if } t_i \equiv auth(s, a, o) \\ & \text{w.r.t. some } (\Sigma_P, P, auth) \\ compute(t_i, C') & \text{if } t_i \equiv comb'(t'_1, \ldots, t'_m) \\ & \text{w.r.t. some } (\Sigma_{C'}, C', comb') \end{cases}$$

$(\mathbf{r_1})$ $auth_{D_1}(p(n(X)), read, rec(n(Z))) \mapsto$
$case(DL(\mathcal{K}_1, instance(X, nurse_{D1}))$ and $DL(\mathcal{K}_2, instance(Z, patient_{D1})) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_2})$ $auth_{D_2}(p(n(X)), read, rec(n(Z))) \mapsto$
$case(DL(\mathcal{K}_1, instance(X, nurse_{D2}))$ and $DL(\mathcal{K}_2, instance(Z, patient_{D2}))) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_3})$ $auth_A(p(n(X)), Y, rec(n(Y))) \mapsto$
$case(DL(\mathcal{K}_3, instance(X, employee_A))$ and $Y = read$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D1} \sqcup patient_{D2}))) \Rightarrow permit$
$case(X = deptChief(dep_A)$ and $Y = write$ and
$DL(\mathcal{K}_2, instance(Z, patient_{D1} \sqcup patient_{D2}))) \Rightarrow permit$
$case(true) \Rightarrow deny$
$(\mathbf{r_4})$ $deptChief(Dep) \mapsto head(DL(\mathcal{K}_3, \exists isChief.\{Dep\}))$

Fig. 3. Policy rules of Example 7

Example 7. Consider a healthcare domain consisting of an administrative department A, and two surgery departments D_1 and D_2. Each department X is modeled as an individual dep_X. Suppose that nurses working in D_1 (resp. D_2) are only allowed to read medical data of patients in D_1 (resp. D_2). Administrative employees of A are allowed to read medical data of any patient, and the chief of A is allowed to write medical data of any patient. Access control policies for D_1, D_2 and A might be specified by using policy rules of Figure 3[4]; specifically, they can be formalized by $\mathcal{D}_1 \equiv (\Sigma_{D_1}, \{r_1\}, auth_{D1})$, $\mathcal{D}_2 \equiv (\Sigma_{D_2}, \{r_2\}, auth_{D2})$, and $\mathcal{A} \equiv (\Sigma_A, \{r_3, r_4\}, auth_A)$, respectively.

Now, suppose that *jane* is a nurse working in D_1 with some administrative duties in A. If *jane* wants to read some medical data about patient *charlie* belonging to D_2, policy $\mathcal{A}$ will permit it, while policy $\mathcal{D}_2$ will not. Since *jane* needs to read such medical data to perform her administrative duties, the permit-override policy combiner can be used to solve the conflict. In particular, the composition of policies $po(auth_A(p(n(jane)), read, rec(n(charlie))), auth_{D2}(p(n(jane)), read, rec(n(charlie))))$ derives the decision permit.

Policy Delegation. Authorization systems quite commonly support *permission delegation* (see [19,11]), that is, an identified subject in the authorization system provided with some permissions can delegate (a subset of) its permissions to another identifiable subject (or group of subjects). In our framework, such feature can be implemented as follows. Suppose that the subject s_1, whose permissions are defined by the access control policy $\mathcal{P}_1 \equiv (\Sigma_1, P_1, auth_1)$, wants to delegate subjects in the set S_d to perform actions in the set A_d over objects in the set O_d. Permission delegation can be formalized by a policy $\mathcal{P}$ containing rules of the form $auth_P(s, a, o) \mapsto case(\langle delegation\_constraint \rangle) \Rightarrow auth_1(s_1, a, o)$, where *delegation_constraint* is a condition that allows us to check whether s,a, and o belongs to the delegation sets S_d, A_d, and O_d. To avoid interferences between

[4] We assume that $\mathcal{K}_1$, $\mathcal{K}_2$, and $\mathcal{K}_3$ are knowledge bases modeling our distributed healthcare domain.

$$
\begin{array}{l}
(\mathbf{r_1^*})\ auth_{D_1}(p(n(X)), Y, rec(n(Z))) \mapsto \\
\quad case(DL(\mathcal{K}_1, instance(X, nurse_{D1}))\ \text{and}\ Y = read\ \text{and} \\
\quad\quad DL(\mathcal{K}_2, instance(Z, patient_{D1})) \Rightarrow permit \\
\quad case(X = patty\ \text{and}\ Y = write\ \text{and}\ DL(\mathcal{K}_2, instance(Z, patient_{D1})) \Rightarrow \\
\quad\quad auth_A(p(n(deptChief(dep_A))), write, rec(n(Z))) \\
\quad case(true) \Rightarrow deny
\end{array}
$$

Fig. 4. The new authorization policy for D_1 implementing a delegation.

applications of rules of $\mathcal{P}$ and $\mathcal{P}_1$, we assume that every authorization request $auth_1(s, a, o)$ is evaluated using only the policy rules in $\mathcal{P}_1$.

Example 8. Consider the healthcare domain of Example 7. Suppose that the chief of department A wants to delegate *patty*, a nurse working in $D1$, to write medical data of patients in $D1$. We can formalize such a delegation by replacing rule r_1 by the new rule r_1^* shown in Figure 4. Roughly speaking, rule r_1^* extends rule r_1 by specifying that *patty* will inherit the chief authorization whenever she wants to write medical data of patients in department $D1$.

Policy Closure. Policy closure operators allow one to infer decisions for a subject s by analyzing decisions for subjects that are semantically related to s, (e.g. if an employee can read a document, then her boss will). In our framework, such operators can be naturally encoded by exploiting semantic relations conveyed by DL roles. The basic idea is as follows. Consider a role R connecting two subjects s, s' by means of the role assertion $R(s, s')$, and an access control policy $\mathcal{P}$ modeling the access privileges for s' w.r.t. an action a and an object o. If $\mathcal{P}$ derives the decision d w.r.t. (s', a, o), then we infer the same decision d for the subject s. This inference scheme works fine whenever the role R models a *one-to-one* semantic relations (i.e. an injective function). Indeed, when R specifies a *one-to-many* relation between subjects decision conflicts may arise, since several distinct decisions might be computed for distinct subjects s' which are semantically related to s. In this case, a conflict-resolution criterion is needed to infer a decision for subject s. More formally, policy closures are defined as follows.

$$
\begin{array}{rl}
clo(s,a,o) \mapsto & \text{if } (DL(\mathcal{K}, \exists R.\{s\})) = [\,] \text{ then } NotApp \\
& \text{else } comb_C(applyPol(triples(a, o, DL(\mathcal{K}, \exists R.\{s\})))) \\
applyPol([\,]) \mapsto & [\,] \\
applyPol((s,a,o) : ts) \mapsto & auth_P(s,a,o) : applyPol(ts) \\
triples(a,o,[\,]) \mapsto & [\,] \\
triples(a,o,x : xs) \mapsto & (x,a,o) : triples(a,o,xs)
\end{array}
$$

Fig. 5. Policy closure rules of Definition 8

Definition 8. *Let $\mathcal{K}$ be a knowledge base and R be a role in $\mathcal{K}$, $\mathcal{P} \equiv (\Sigma_P, P, auth_P)$ be a policy and $\mathcal{C} \equiv (\Sigma_C, C, comb_C)$ be a policy combiner. A* closure *of $\mathcal{P}$ w.r.t. R and $\mathcal{C}$, is a policy $\mathcal{PC} \equiv (\Sigma_{PC}, PC, clo)$, where Σ_{PC} is the policy signature, PC is the set of policy rules of Figure 5, and $clo \in PC$ is the policy evaluator.*

Basically, evaluating *clo* on a triple (s, a, o) amounts to applying the policy evaluator $auth_P$ on the triples in the set $\{(s_1, a, o), \ldots, (s_n, a, o)\}$, where $R(s, s_i)$

holds in $\mathcal{K}$ for all $i \in \{1, \ldots, n\}$. This operation leads to a set of decisions $\{d_1, \ldots, d_n\}$, which are combined together according to the chosen combination operator $comb_C$. The final result of this process is a single decision corresponding to the evaluation of the closure of the policy $\mathcal{P}$ w.r.t. R and $\mathcal{C}$. To avoid interferences between applications of rules of $\mathcal{P}$ and $\mathcal{C}$, we assume that every authorization request $auth_P(s_i, a, o)$ is evaluated using only the rules in $\mathcal{P}$, while the combination $comb_C(d_1, \ldots, d_n)$ is evaluated using the policy rules in $\mathcal{C}$.

Example 9. Consider the access control policy $\mathcal{P}_\mathcal{H}$ specified in Example 3 where only designated physicians may write patient medical records. Assume that the knowledge base $\mathcal{H}$ also contains the role *supervises*, which intuitively specifies that fact that some physician may supervise multiple (junior) physicians. Now, we would like to formalize that physicians supervising at least one junior physician can write patient medical records, even if they are not designated. To this end, it suffices to construct the closure of policy $\mathcal{P}_\mathcal{H}$ w.r.t. the role *supervises* and the policy combiner *po* specified in Example 6.

5 Checking Domain Properties of Access Control Policies

In our framework, Description Logic is employed to model the domains to which a given access control policy is applied: subjects, actions, objects, as well as relations connecting such entities can be specified via DL knowledge bases. Therefore, the structure of the policy domains can be naturally analyzed by means of DL reasoning services. More specifically, the idea is to formalize properties over the domains of interest by means of policy rules. Then, the d-rewriting mechanism can be applied to verify the specified properties.

Definition 9. *Let Σ_S be a policy signature. A* domain property specification *of properties $p_1, \ldots, p_n$ is a triple $(\Sigma_S, S, \{p_1, \ldots, p_n\})$, where S is a set of policy rules, and $p_1, \ldots, p_n$ are terms in $\tau(\Sigma_S)$ such that each p_i is an instance of a lhs of some policy rule in S.*

Example 10 below shows that domain property specifications are expressive enough to formulate several well known policy constraints such as separation of duty, cardinality constraints, *etc.*

Example 10. Let $\mathcal{H}$ be the knowledge base of Example 1 modeling the policy domain of the policy specified in Example 3. The following properties
– **sep_of_duty.** No guardian can be a physician.
– **at_most_4.** A physician can be assigned at most to four patients.
can be specified by the domain property specification
$(\Sigma_H, S_H, \{sep\_of\_duty(physician, guardian), at\_most(4)\})$
such that S_H contains
– $sep\_of\_duty(X, Y) \mapsto DL(\mathcal{H}, subsumes(\neg X, Y))$
– $at\_most(X) \mapsto DL(\mathcal{H}, subsumes(\bot, (\geqslant_{X+1} assignedTo^-) \sqcap physician)))$
and Σ_H is a policy signature including all the symbols occurring in S_H.

In this context, verifying a domain property p amounts to finding a finite d-rewrite sequence which reduces p to the boolean value *true*.

Definition 10. *Let $\mathcal{S} \equiv (\Sigma_S, S, \{p_1, \ldots, p_n\})$ be a domain property specification of properties $p_1, \ldots, p_n$. Then, p_i holds in S iff $p_i \mapsto_S^+ true$.*

6 Conclusions

Domain specific languages play a key role in access control; since, on the one hand, they allow security administrators to formally specify precise policy behaviors; and on the other hand, formal methods can be applied giving support to both analysis and verification of access control policies. In this paper, we proposed a novel rule-based language which is particularly suitable for managing security in the semantic web, where access control information may be shared across multiple sites and depends on the semantic descriptions of the resources to be protected. We have also shown that semantic metadata can be exploited to specify and check properties related to the considered policy domain.

The proposed action control language has been implemented in the prototype system Paul, which is written in the functional language Haskell, and whose source code is freely available at [3]. The d-rewriting evaluation mechanism is built around the Haskell's lazy evaluation mechanism. Basically, we integrated DL reasoning capabilities into such an engine by using the DIG interface [5], which is an XML standard for connecting applications to remote DL reasoners[5]. The DIG interface is capable of expressing the description logic formalized within the OWL-DL [25] framework (namely, $\mathcal{SHOIQD}_n^-$ logic). Therefore, our system fully exploits both the efficiency of Haskell and the reasoning power of $\mathcal{SHOIQD}_n^-$ logic, providing fast evaluations of authorization requests. To evaluate the expressiveness and efficiency of our language, we tested some access control policy specifications, which are available at Paul's web site [3]. As future work, we intend to potentiate the verification capabilities of our framework by developing narrowing-based analyses in the style of [17]. Hopefully, this will give support to policy repair and optimization techniques.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Scheider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. M. Baggi, D. Ballis, and M. Falaschi. Paul - the Policy Specification and Analysis Language, 2009. URL: `http://sole.dimi.uniud.it/~michele.baggi/paul`.
4. S. Barker and M. Fernández. Term Rewriting for Access Control. In *Proc. of DBSec '06*, pp. 179–193. Springer LNCS 4127, 2006.
5. S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. Technical report, University of Manchester, 2003.

[5] In our experiments, we have used Pellet [22], an efficient, open-source DL reasoner for the OWL-DL framework.

6. C. Bertolissi and M. Fernández. A Rewriting Framework for the Composition of Access Control Policies. In *Proc. of PPDP '08*, pages 217–225. ACM, 2008.
7. P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An Algebra for Composing Access Control Policies. *ACM TISS*, 5(1):1–35, 2002.
8. E. Damiani, S. D. C. di Vimercati, C. Fugazza, and P. Samarati. Extending Policy Languages to the Semantic Web. In *Proc. of ICWE'04*, pp. 330–343. Springer LNCS 3140, 2004.
9. M. Dean and G. Schreiber. OWL Web Ontology Language Reference — W3C recommendation, 2004. URL: `http://www.w3.org/TR/owl-ref/`.
10. G. Denker, L. Kagal, T. W. Finin, M. Paolucci, and K. P. Sycara. Security for DAML Web Services: Annotation and Matchmaking. In *Proc. of ISWC'03*, pp. 335–350. Springer LNCS 2870, 2003.
11. J. DeTreville. Binder, a Logic-Based Security Language. In *Proc. of IEEE SSP'02*, pages 105–113. IEEE Computer Society, 2002.
12. D. J. Dougherty, C. Kirchner, H. Kirchner, and A. S. de Oliveira. Modular Access Control Via Strategic Rewriting. In *Proc. of ESORICS '07*, pp. 578–593. Springer LNCS 4734, 2007.
13. T. W. Finin and A. Joshi. Agents, Trust, and Information Access on the Semantic Web. *SIGMOD Record*, 31(4):30–35, 2002.
14. R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett. No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web. In *Proc. of ESWS'04*, pp. 342–356. Springer LNCS 3053, 2004.
15. L. Kagal, T. Berners-Lee, D. Connolly, and D. J. Weitzner. Using Semantic Web Technologies for Policy Management on the Web. In *Proc. of AAAI'06*. AAAI Press, 2006.
16. L. Kagal, T. W. Finin, and A. Joshi. A Policy Based Approach to Security for the Semantic Web. In *Proc. of ISWC'03*, pp. 402–418. Springer LNCS 2870, 2003.
17. C. Kirchner, H. Kirchner, and A. S. de Oliveira. Analysis of Rewrite-Based Access Control Policies. *ENTCS*, 234:55–75, 2009.
18. V. Kolovski, J. Hendler, and B. Parsia. Analyzing Web Access Control Policies. In *Proc. of WWW '07*, pp. 677–686. ACM, 2007.
19. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM TISS*, 6(1):128–171, 2003.
20. T. Moses. eXtensible Access Control Markup Language (XACML) v2.0. Technical report, OASIS, 2005.
21. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
22. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: a Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.
23. A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and J. S. Aitken. Kaos Policy Management for Semantic Web Services. *IEEE Intelligent Systems*, 19(4):32–41, 2004.
24. World Wide Web Consortium (W3C). Web Services Policy 1.2 - framework (WS-Policy), 2006. URL: `http://www.w3.org/Submission/WS-Policy/`.
25. World Wide Web Consortium (W3C). OWL Web Ontology Language Guide, 2004. URL: `http://www.w3.org/TR/owl-guide/`.
26. C. Zhao, S. L. N. Heilili, and Z. Lin. Representation and Reasoning on RBAC: A Description Logic Approach. In *Proc. of ICTAC '05*, pp. 381–393. Springer LNCS 3722, 2005.

Lazy and Faithful Assertions for Functional Logic Programs

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. Assertions or contracts are an important technique to improve the quality of software. Thus, assertions are also desirable for functional logic programming. Unfortunately, there is no established meaning of assertions in languages with a demand-driven evaluation strategy. Strict assertions are immediately checked but may influence the behavior of programs. Lazy assertions do not modify the behavior but may not be faithful since some assertions might not be checked at all. This paper proposes an intermediate approach where the user can choose between lazy or faithful assertions. In order to avoid disadvantages of faithful assertions, we propose to delay them until an explicit point where faith is required, e.g., at the end of the program execution or before I/O actions. We describe a prototypical implementation of this idea in the functional logic language Curry.

1 Motivation

The use of assertions or contracts is an important technique to improve the quality of software [20]. Assertions make certain assumptions in the code explicit, e.g., requirements on argument values to ensure the correct execution of a function's code. In principle, assertions can be implemented in the program's code by including code to check them. For instance, one can raise an exception if the factorial function is called with a negative argument or the `head` function is applied to an empty list. In order to keep the application code comprehensible and maintainable, it is preferable to have a clear distinction between application code and assertions so that one can later decide how to treat assertions. For instance, they can be always checked, checked only during the development and test of the application program, or removed after proving that they hold in the application program.

Design by contract has been introduced in the context of object-oriented programming [20] but their use in other programming paradigms is also reasonable. In this paper we consider the inclusion of assertions[1] in functional logic programs. Thus, we assume familiarity with basic concepts of functional logic programming

[1] We use the term "assertions" for properties of values, whereas "contracts" are used for properties of operations consisting of assertions for arguments as well as assertions for result values.

(details can be found in a recent survey [14]). For our examples and implementation, we use the declarative multi-paradigm language Curry [11,16] that combines functional programming features (demand-driven evaluation, higher-order functions) with logic programming features (computing with partial information, unification, non-deterministic search for solutions) and concurrent evaluation. The syntax of Curry is almost identical to Haskell [21]. In addition to Haskell, Curry allows the declaration of free (logic) variables by the keyword "`free`".

The intuitive meaning of assertions is as follows. If we decorate an expression in a program with an assertion, e.g., a predicate, then an assertion violation should be reported whenever, during the program's execution, the expression has some value that does not satisfy the assertion. Unfortunately, a precise definition and implementation of assertions is not straightforward in the context of functional logic programming due to the demand-driven evaluation strategy. This problem is already present in non-strict functional languages where various proposals have been made to tackle it (e.g., [5,6,7]). In order to discuss the difficulties in more detail, consider a simple approach to introduce assertions in a functional (logic) language by defining a combinator that attaches an assertion to an expression:

```
assert :: (a -> Bool) -> a -> a
assert p x = if p x then x
                    else error "Assertion failed"
```

Here, the assertion is a predicate on the values of the expression. If the predicate applied to the expression evaluates to `True`, the expression is returned, otherwise an exception is raised. Since this definition has the effect that the assertion is immediately checked, we call such an assertion also *strict assertion*.

A disadvantage of strict assertions is the fact that they are not *meaning preserving*, i.e., they might influence the behavior of application code, even if all assertions are satisfied. For instance, consider an assertion that states that a list is ordered:

```
ordered []       = True
ordered [_]      = True
ordered (x:y:ys) = x<=y && ordered (y:ys)
```

Then the evaluation of "`head (assert ordered [1,2..])`" does not terminate due to the evaluation of the infinite list argument caused by the assertion.

To avoid this influence of assertions to the application program, Chitil et al. [7] proposed *lazy assertions* that do not enforce argument evaluation but are checked when the argument expression has been evaluated by the application program so far that the assertion can be evaluated without further evaluation of its argument. Thus, as long as all assertions are satisfied, program executions with or without lazy assertion checking deliver the same results. A disadvantage of lazy assertions is the fact that some obviously violated assertions are not reported when the arguments are not sufficiently evaluated. For instance, "`head (assert ordered [2,1])`" returns `2` without any assertion violation if the assertion is lazily checked, although

the programmer assumes, due to the intuitive meaning of assertions, that the result is the minimal element of the list.

Chitil and Huch [5,6] improved the situation by introducing a specific assertion language that supports the "prompt" evaluation of assertions. Nevertheless, the basic problem remains: it is possible that the violation of assertions might be undetected or detected too late if all assertions are lazily checked. One can argue that this behavior is fine since the possibly unchecked parts of an expression are not necessary for computing the result. However, this view changes when I/O actions are taken into account. For instance, if we pass a data structure through a sequence of I/O actions where a first action needs only some part of the structure (similarly to the call to `head` above) and the other parts are needed by subsequent actions, then a violation of an assertion for the data structure might be detected too late, e.g., after some rocket has been launched. Thus, there are situations where one wants to ensure that the assertions hold. Degen et al. [8] put this into the slogan "faithfulness is better than laziness."

Altogether, there is no silver bullet for assertions in non-strict languages since lazy assertions might not be faithful, and strict assertions cannot be applied to algorithms exploiting infinite data structures. Thus, we propose an intermediate approach where the user can choose between lazy and faithful assertions. In contrast to purely functional languages, we propose to evaluate even faithful assertions in a lazy manner but enforce their evaluation at particular program points, e.g., before I/O actions where faith is important or at the end of the program execution.

One might wonder why it is useful to evaluate faithful assertions not simply as strict assertions but treat them in a lazy manner. The reason is that this strategy can reduce the possibility to report false violations of assertions, i.e., violations that do not occur during the execution of the application program. Note that lazy evaluation in functional *logic* programs is not only desirable to obtain optimal evaluation [2] but also to reduce the search space, i.e., lazy evaluation on non-deterministic programs yields a demand-driven exploration of the search space (compare [1,14]). As a consequence, it is reasonable to evaluate expressions lazily even if we know that we want to evaluate them completely. For instance, consider the program

```
f 0 = 0
f 1 = 1

g x = [f x]

h False 1 = 2
h True  0 = 3
```

together with the expression "`let x free in h (null (g x)) x`" (where the predefined operation `null` checks whether its argument is the empty list). Functional logic languages with a lazy evaluation strategy, like Curry [11,16] and $\mathcal{TOY}$ [18], evaluate this expression to `2` by instantiating `x` to `1` (after evaluating `(null (g x))` to `False`). Now consider that we put a post-condition on `g` ensur-

ing that all values in the result list are positive whenever we apply `g` to some argument during the program execution, i.e., we modify the definition of `g` to

```
g x = assert (all (>0)) [f x]
```

If the assertion is eagerly evaluated when the evaluation of `g` is demanded, `f x` is evaluated to `0` by instantiating `x` to `0` so that an exception is raised due to the violated assertion. However, if we delay the assertion checking until the end of the regular program execution (which instantiates `x` to `1`), no exception will be raised since the assertion is satisfied.

Thus, we propose in this paper an assertion framework for functional logic programs with the following characteristics:

- The programmer can put lazy or faithful assertions on expressions in the application program.
- Lazy assertions are checked when the arguments of the assertions are evaluated by the application program so that the assertions can be reduced to a Boolean value without any further evaluation of the arguments. Thus, lazy assertions do not initiate any argument evaluation by themselves.
- Faithful assertions are also lazily checked, i.e., when the values of their arguments are available, they are reduced to a Boolean value. In addition, the programmer can specify execution points (I/O actions) where all faithful assertions are eagerly checked if they have not been already checked.
- We describe a prototypical implementation of the framework in Curry. The implementation is defined as a library implemented in Curry based on a few extensions available in the Curry implementation PAKCS [15].

The next section defines the kind of assertions which we propose in this paper. Section 3 describes an implementation of our assertion concept in Curry. Section 4 discusses some related work before we conclude in Section 5.

2 Assertions

As discussed in Section 1, assertions can be considered as predicates on expressions, i.e., they are of type "`a -> Bool`" where `a` is the type of the considered expression. Of course, one can deal with richer and more specialized assertion languages, like [5,6,17], but this is outside the scope of this paper. Therefore, we simply consider assertions as standard predicates. As shown in the previous section, one could attach an assertion to an expression by a combinator

```
assert :: (a -> Bool) -> a -> a
```

However, this is not sufficient for an implementation of lazy assertions since they need to inspect the data on which they operate. Therefore, they are not parametrically polymorphic but their behavior depends on the structure of the concrete type. Hence, we adopt a technique used in observation debugging tools for functional (logic) languages [4,10] and put some information about the considered

types as an additional argument of type "`Assert a`".[2] Furthermore, we also add a string argument that is used to identify the violated assertion when an exception is raised.[3] Altogether, the assertion combinator has the following type:

```
assert :: Assert a -> String -> (a -> Bool) -> a -> a
```

In order to attach concrete assertions to a program, our assertion library defines constants and functions returning assertion information for particular types, like

```
aInt   :: Assert Int
aFloat :: Assert Float
aChar  :: Assert Char
...
aList  :: Assert a -> Assert [a]
aPair  :: Assert a -> Assert b -> Assert (a,b)
...
```

A concrete assertion is defined by combining these operations in a type correct way. For instance, the post-condition on the operation `g` as shown in Section 1 can now be defined by

```
g x = assert (aList aInt) "AllPositive" (all (>0)) [f x]      (1)
```

These assertions are *faithful*, i.e., they are lazily evaluated with the same demand as the application program but they can also be strictly checked at particular program points. For the latter purpose, the library defines an I/O action

```
checkAssertions :: IO ()
```

that forces the evaluating of all pending assertions. Thus, one can check all assertions at the end of the user program (`main`) by executing

```
main >> checkAssertions
```

Of course, `checkAssertions` can be also used any number of times during the regular program execution, e.g., before "important" I/O actions of the user program.

If the programmer wants to define assertions that should be only lazily evaluated (i.e., they might not be faithful but could be desirable for assertions on infinite data structures), our library also provides an operator to attach such *lazy assertions* to expressions:

```
assertLazy :: Assert a -> String -> (a -> Bool) -> a -> a
```

[2] In Haskell one could add the `Assert` information via type classes, but type classes are not yet included in Curry.

[3] Of course, it would be better to show the position of the violated assertion in the exception. Since this information cannot be obtained by a library but require specific compiler support, we omit it here. In a future version, the compiler might introduce the position information in the string argument.

In order to test and compare the various assertion methods, our library also defines a *strict assertion*, which is immediately checked, by

```
assertStrict :: Assert a -> String -> (a -> Bool) -> a -> a
assertStrict _ id p x =
   if p x then x
          else error ("Strict assertion '"++id++"' failed!")
```

Consider again the example given in Section 1 where the function `g` is defined with a post-condition as shown in program rule (1) above. Our implementation, described in detail below, evaluates the expression "`h (null (g x)) x`" without reporting an assertion violation, as intended. However, if we replace in rule (1) "`assert`" by "`assertStrict`", the evaluation of the same expression yields an exception reporting that the assertion `AllPositive` is violated, which is not true in the program executed without assertions. This example shows the usefulness to evaluate faithful assertions in a lazy manner. The next section shows an implementation of this concept in Curry.

3 Implementation

In this section, we first describe an implementation of purely lazy assertions in Curry. Based on this, we develop an implementation of faithful assertions.

3.1 Lazy Assertions

An implementation of lazy assertions in Haskell has been proposed in [7]. Our implementation[4] uses similar ideas but is based on functional logic programming features. Lazy assertions should only be checked when the application program demands and evaluates the arguments of the assertions. In order to avoid the evaluation of arguments by assertion checking, we wrap these arguments by a function `wait` that is evaluable only if the original argument has been evaluated by the application program. In [7] this is implemented via concurrent threads which synchronize on `IORefs`. Since Curry subsumes the concept of concurrent logic programming, we use these features to implement the concurrent evaluation of lazy assertions.

To implement lazy assertions, we need for each concrete type "`a`" two operations:

```
wait    :: a -> a
ddunify :: a -> a -> a
```

`wait` is the identity function on values of type `a` but suspends as long as the value is not provided.[5] For instance, consider the type `Nat` of natural numbers in Peano's notation defined by

[4] This implementation is partially based on code developed with Bernd Braßel and Olaf Chitil.

[5] The suspension is important to ensure the principle that lazy assertions should not change the evaluation behavior of the application program.

```
data Nat = Z | S Nat
```

The corresponding operation `waitNat` can be defined as follows:

```
waitNat :: Nat -> Nat
waitNat x = case x of Z   -> Z
                      S y -> S (waitNat y)
```

Clearly, `waitNat` is the identity on `Nat` values but suspends when it is applied to a free variable (due to the `case` construct which suspends on free variables, see [16]).

The second important operation, `ddunify`, implements a demand-driven unification of its arguments. Conceptually, a call "`ddunify x e`" evaluates `e` (demand-driven, i.e., to head-normal form), returns the result, and unifies `x` (which is usually a free variable) with the result's top-level constructor and recurs on the arguments. For instance, the corresponding operation for the type `Nat` is defined as follows:

```
ddunifyNat :: Nat -> Nat
ddunifyNat x e =
 if isVar e then (x=:=e) &> e
            else case e of
                   Z   -> (x=:=Z) &> Z
                   S y -> let z free
                          in (x=:=S z) &> S (ddunifyNat z y)
```

The test function `isVar` checks whether the current argument evaluates to a free variable (note that free variables are also head-normal forms in functional logic programs). Although this test function is non-declarative (it has been introduced in [4] for a similar purpose), it is necessary to check the state of the argument in order to avoid its unintended instantiation. If the argument `e` evaluates to a free variable, it is unified (by the equational constraint "`=:=`") with the argument `x` and returned.[6] Otherwise, the possible constructor-rooted values are examined. In case of the constructor `Z`, this constructor is unified with argument `x` and returned. If the argument's value is rooted by the constructor `S`, `x` is instantiated to the constructor `S` with a fresh variable argument and the corresponding arguments are further unified by `ddunifyNat`.

As we will see below, the operations `wait` and `ddunify` are sufficient to implement lazy assertions. Thus, the type `Assert` to encapsulate type-specific assertion information is defined as

```
data Assert a = Assert (a -> a) (a -> a -> a)
```

where the first and second component are the type-specific `wait` and `ddunify` operations, respectively. Thus, an instance of `Assert` for the concrete type `Nat` can be defined by

```
aNat :: Assert Nat
```

[6] "`c &> e`" denotes a *guarded expression* where the infix operator is predefined by the conditional rule "`c &> e | c = e`".

```
aNat = Assert waitNat ddunifyNat
```

Based on this structure of the type `Assert`, we can implement the combinator `assertLazy` by applying the operations passed with the `Assert` argument:

```
assertLazy ::  Assert a -> String -> (a -> Bool) -> a -> a
assertLazy (Assert wait ddunify) label p e =
  spawnConstraint (check label (p (wait x))) (ddunify x e)
 where x free
```

The operation `spawnConstraint` (first introduced in [4]) is identical to the guarded expression operator "`&>`" from a declarative point of view. In contrast to "`&>`", `spawnConstraint` proceeds with the evaluation of the second argument even if the evaluation of the guard (first argument) suspends (i.e., the guard is concurrently evaluated). Thus, `assertLazy` evaluates its expression argument `e` and unifies its value with the free variable `x` in a demand-driven manner (by "`ddunify x e`"). Concurrently, the assertion `p` is applied to `x` wrapped by the operation `wait` in order to delay the evaluation until the required argument value is available. The operation `check` simply examines the result of assertion checking and raises an exception, if necessary:

```
check :: String -> Bool -> Success
check label result =
  case result of
    True  -> success
    False -> error ("Lazy assertion '"++label++"' violated!")
```

Due to the use of the operations `wait` and `ddunify`, the behavior of the computation of the application program is not changed by attaching lazy assertions to expressions. Since spawned constraints are evaluated with a high priority, a violated assertion is reported as soon as it can be decided by the availability of the argument values.

Before we discuss the implementation of faithful assertions, we discuss further instances of `Assert` for some concrete types. One can define an instance for the primitive type of integers (and similarly for other non-structured types) by

```
aInt :: Assert Int
aInt = Assert waitInt ddunifyInt
 where waitInt = ensureNotFree
       ddunifyInt x i | x=:=i = i
```

The predefined operation `ensureNotFree` returns its argument evaluated to head normal form but suspends as long as the result is a free variable. The application of `ensureNotFree` avoids an unintended instantiation of free variable arguments during the evaluation of an assertion.

Instances of `Assert` for polymorphic type constructors take assertion information related to their argument types. For instance, the combinator `aList` is defined as follows:

```
aList :: Assert a -> Assert [a]
aList (Assert waita ddunifya) = Assert waitList ddunifyList
 where
  waitList l = case l of
                 []     -> []
                 (x:xs) -> waita x : waitList xs

  ddunifyList x e =
   if isVar e
   then (x=:=e) &> e
   else
    case e of
     []   -> (x=:=[]) &> []
     y:ys -> let z,zs free
              in (x=:=z:zs) &> (ddunifya z y : ddunifyList zs ys)
```

We have shown the implementation of `Assert` instances for a few types. Note, however, that the code structure of these instances follows a scheme which depends on the structure of the type definitions. Thus, it is easy to provide an automatic tool to generate these instances for any user-defined data type.

3.2 Faithful Assertions

As mentioned above, faithful assertions are evaluated like lazy assertions during standard execution of the application program. In addition, they are eagerly evaluated when it is requested by the I/O action `checkAssertions`. This demands for two execution modes of faithful assertions:

1. demand-driven evaluation like lazy assertions, and
2. eager evaluation like strict assertions.

The implementation of lazy assertion wraps the arguments via `wait` operations as shown above, i.e., their values are not available to the assertion as long as they are not demanded by the application program. Thus, there seems to be no way to enforce the evaluation of a lazy assertion outside the application program. Due to this consideration, we implement the eager evaluation mode of a faithful assertion by creating a further application of the assertion to its original arguments. The evaluation of this application is delayed until it is requested by `checkAssertions`. Obviously, this scheme could cause some re-evaluation of the assertion when the evaluation of the lazy assertion has already started before `checkAssertions` occurs. However, this is not a serious problem since the evaluation of the arguments are shared (due to the lazy strategy of the host language) and under the assumption that assertion evaluation has not a high complexity (which should be satisfied in order to execute programs with assertion checking in a reasonable amount of time).

At least, we can avoid the re-evaluation of an already evaluated assertion by passing a free variable as an "evaluation flag." Thus, we extend the implementation of lazy assertions shown in Section 3.1 to faithful assertions as follows:

```
assert ::  Assert a -> String -> (a -> Bool) -> a -> a
assert (Assert wait ddunify) label p e
  | registerAssertion eflag label (p e)
  = spawnConstraint (check label eflag (p (wait x))) (ddunify x e)
 where eflag,x free
```

The operation `registerAssertion` suspends the evaluation of its last argument until it is requested by `checkAssertions`. The free variable `eflag` is the flag to avoid a complete re-evaluation of the assertion. For this purpose, we redefine the function `check` presented above so that the variable `eflag` is instantiated when the assertion is fully evaluated (the instantiation in the `False` case is reasonable if the exception is caught):

```
check label eflag result = case result of
  True  -> eflag=:=()
  False -> eflag=:=() &>
           error ("Lazy assertion '"++label++"' violated!")
```

Next we discuss the implementation of `registerAssertion`. Suspended computations can be easily obtained in Curry by waiting on the instantiation of a free variable. Since all faithful assertions should be evaluated if `checkAssertions` occurs, these suspended computations must share the same free variable. Therefore, we use the Curry library `GlobVar` that supports the definition of typed "global variables." A global variable is a top-level entity that has an associated term. Furthermore, the association can be changed by I/O actions. A global variable `g` having associated terms of type `t` is defined in a Curry program by

```
g :: GlobVar t
g = globVar v
```

where `v` is the initially associated term (of type `t`). Furthermore, there are two operations

```
getGlobVar :: GlobVar a -> IO a
setGlobVar :: GlobVar a -> a -> IO ()
```

to get and set the term associated to a global variable. Note that the associated term can also contain free variables.

We define a global variable that is associated to a free variable used to synchronize all faithful assertions:[7]

```
assertionControl :: GlobVar ()
assertionControl = globVar unknown
```

Using this global variable, we can define the operation to register a faithful assertion for later evaluation as follows:

```
registerAssertion eflag label asrt =
```

[7] The operation `unknown` is defined by "`unknown = x where x free`" in the Curry prelude, i.e., it returns a free variable.

```
  spawnConstraint (delayedAssertion eflag label asrt =:= ())
                  success

delayedAssertion eflag label asrt = unsafePerformIO $ do
  v <- getGlobVar assertionControl
  ensureNotFree v =:= ()  &> done
  if isVar eflag && not asrt
   then error ("Faithful assertion '"++label++"' violated!")
   else done
```

Since global variables are handled by I/O actions, we have to use the operation `unsafePerformIO` to put the execution of an I/O action into a non-I/O value. Hence, `registerAssertion` spawns a constraint which waits on the instantiation of the variable that controls faithful assertions. If this variable is instantiated, it is checked whether the evaluation flag is instantiated, i.e., whether the corresponding lazy assertion was already evaluated. If this is not the case, the assertion `asrt` is evaluated and an exception is raised in case of a violation.

Now it is easy to request that the evaluation of faithful assertions by instantiating the global control variable:

```
checkAssertions = do
  v <- getGlobVar assertionControl
  v=:=() &> done           -- instantiate control variable
  setGlobVar assertionControl unknown
```

After the instantiation, a new free variable is associated to the global control variable in order to have it ready for subsequent occurrences of faithful assertions.

Note that we used a few non-declarative features to implement lazy and faithful assertions. This is not a problem for the application programmer since their use is completely hidden in the library implementing this assertion framework. The use of the non-declarative constructs is quite helpful to obtain a maintainable high-level implementation without extending the run-time system of the underlying Curry implementation.

3.3 More Assertions

We can use the assertion implementation to provide some other useful assertions. For instance, an assertion for a function could check whether all pairs of argument/result values satisfy a given predicate:

```
assertFun :: Assert a -> Assert b -> String -> (a->b->Bool)
             -> (a->b) -> a -> b
assertFun (Assert waita ddunifya) (Assert waitb ddunifyb)
          label p f x
  | registerAssertion eflag label (p x (f x))
  = spawnConstraint (check label eflag (p (waita wx) (waitb wfx)))
                    (ddunifyb wfx (f (ddunifya wx x)))
 where eflag,wx,wfx free
```

For instance, in order to check whether a function `f` of type `Int -> Int` behaves monotonically for all calls, we wrap it with the assertion

```
assertFun aInt aInt "monotonic" (<) f
```

The notion of *contracts* is usually stronger [20] since it consists of a precondition (on the argument) and a postcondition (relating the argument and the result) for an operation. Thus, we can define a contract as an assertion on the argument and an assertion between argument and result values as follows:

```
contract :: Assert a -> Assert b -> String
            -> (a->Bool) -> (a->b->Bool) -> (a->b) -> a -> b
contract asrta asrtb label argp funp f x =
  assertFun asrta asrtb ("Result of "++label) funp f
            (assert asrta ("Argument of "++label) argp x)
```

For instance, if we want to turn a function `fac` into a function `cfac` containing the contract ensuring that `fac` is always called with a non-negative argument and returns a positive argument, we define it as follows:

```
cfac = contract aInt aInt "fac" (>=0) (\_ r->r>0) fac
```

Sometimes one is interested to ensure that specific arguments are free variables when they occur for the first time. For instance, there are high-level libraries for GUI or HTML programming that use free variables as logical references between widgets and event handlers [12,13]. Although these libraries use abstract data types in order to ensure this property, it might be also useful to check this property by an assertion, in particular, during the development of such libraries. This can be done by an assertion that raises an exception when the argument is not a free variable. The implementation is quite easy (note that faithful assertions are not necessary here since this assertion is immediately checked when it occurs):

```
assertLogVar :: String -> a -> a
assertLogVar label x = (check label () (isVar x)) &> x
```

4 Related Work

Assertions or contracts have been introduced in the context of imperative object-oriented programming languages [20], but it is obvious that assertions are also useful for declarative programming. Although powerful type systems can express assertions on operations that can be checked at compile time, more complex properties, like orderings on lists or trees, non-negative values etc, cannot be expressed by standard type systems. If application programs become more complex, it is important to state and check such properties, e.g., to improve debugging or reliability.

Since the demand-driven evaluation model of functional languages causes additional difficulties when considering assertions, there are a number of different proposals in this area. Chitil et al. [7] proposed lazy assertions for non-strict

functional languages. They suggested that assertions should not influence the normal behavior of programs (apart from space and time needed for assertion checking). For this purpose, they discussed different implementations. To ensure early detection of assertion violations, they implemented assertions by concurrent threads. Although we used the concurrent logic programming features of Curry to implement assertions, both implementations of lazy assertions have many similarities, in particular, both are based on some non-declarative constructs like `unsafePerformIO`.

Since lazy assertions might not detect assertion violations if parts of the considered data structures are not demanded by the application program, Chitil and Huch [5,6] improved the situation by introducing a specific pattern logic to express assertions that allow an earlier detection of violated assertions. In particular, they proposed to replace sequential Boolean operators, like "`&&`" or "`||`", by corresponding operators that evaluate their arguments in parallel. Although such an extended assertion language is interesting in our framework, this does not make our proposal for faithful assertions superfluous: even in the extended assertion language of Chitil and Huch, it might be the case that some violated assertions remain undetected in a program execution.

Degen et al. [8] discussed the various requirements and possibilities of assertion checking in lazy languages and came to the conclusion that there is no method satisfying all desirable requirements. Thus, one has to choose between meaning preserving (i.e., lazy) or faithful (i.e., strict) assertions. We have shown that in functional logic programming, there is a further interesting approach: faithful assertions that are not immediately checked but delayed to a point where faith is strictly required.

Hinze et al. [17] introduced a domain-specific language for defining contracts in Haskell. Contracts are mainly evaluated in an eager manner. Nevertheless, it would be interesting to use their ideas to develop a set of more expressive contract combinators for our framework.

Findler and Felleisen [9] defined a contract system for higher-order functions. In particular, they tackled the problem of correct blame assignment, i.e., to provide precise information about the source position of violated contracts. Although this is orthogonal to the problems addressed in this paper, a correct blame assignment is also relevant in our context and an interesting topic for future work.

Assertions have been also considered in (constraint) logic programming. For instance, [22] proposes a rich assertion language which also includes type and mode information for predicates. [19] combines assertion checking with compile-time verification so that only assertions which cannot be statically verified are dynamically checked. Due to the eager evaluation strategy of Prolog, the difficulties which we address in this paper do not occur there. Nevertheless, it would be interesting to combine our framework with compile-time verification methods.

The demand-driven unification of arguments in our implementation of lazy assertions seems similar to function patterns introduced to obtain more expressive patterns in functional logic programs [3]. In contrast to function patterns, which are completely evaluated to data terms for successful pattern matching,

the demand-driven unification introduced in this paper does not evaluate the expressions completely but is driven by the demand of the evaluation context.

5 Conclusions

We have presented a framework to add lazy and faithful assertions to functional logic programs. Since it is not obvious for any program which properties assertions should satisfy, i.e., whether they should be meaning preserving or faithful, we propose to have both possibilities available in a non-strict language. However, in a functional *logic* language with a demand-driven evaluation strategy, it is reasonable to delay even faithful assertions as long as possible in order to avoid unintended computation branches. We have shown an implementation of this framework in the functional logic language Curry, where we used a few non-declarative constructs to implement the assertion framework as a library without modifying the run-time system. It should be noted that assertions cause some overhead only if they occur in a program. Since we have not modified the run-time system to implement assertions, programs without assertions do not have any overhead due to the availability of assertion checking.

For future work, it would be interesting to consider a more expressive assertion language, like the contract language of [17] or the pattern logic of [5,6]. Furthermore, more work has to be done in order to get a practical tool that automatically provides the source code positions of violated assertions.

Acknowledgements. The author is grateful to the anonymous referees for helpful comments and suggestions to improve this paper.

References

1. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.
4. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 193–208. Springer LNCS 3057, 2004.
5. O. Chitil and F. Huch. Monadic, Prompt Lazy Assertions in Haskell. In *Proc. APLAS 2007*, pp. 38–53. Springer LNCS 4807, 2007.
6. O. Chitil and F. Huch. A Pattern Logic for Prompt Lazy Assertions in Haskell. In *Proc. of the 18th International Symposium on Application and Implementation of Functional Languages (IFL 2006)*, pp. 126–144. Springer LNCS 4449, 2007.
7. O. Chitil, D. McNeill, and C. Runciman. Lazy Assertions. In *Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL 2003)*, pp. 1–19. Springer LNCS 3145, 2004.

8. M. Degen, P. Thiemann, and S. Wehr. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*, pp. 370; 2946–59. Springer LNI 154, 2009.
9. R.B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In *Proceedings of the 7th ACM SIGPLAN international conference on Functional programming (ICFP'02)*, pp. 48–59. ACM Press, 2002.
10. A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr. Notes Theor. Comput. Sci.*, Vol. 41, No. 1, 2000.
11. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
12. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
13. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
14. M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.
15. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2008.
16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
17. R. Hinze, J. Jeuring, and A. Löh. Typed Contracts for Functional Programming. In *Proc. Eight International Symposium on Functional and Logic Programming (FLOPS 2006)*, pp. 208–225. Springer LNCS 3945, 2006.
18. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
19. E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP 2009)*, pp. 281–295. Springer LNCS 5649, 2009.
20. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, second edition, 1997.
21. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
22. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pp. 23–62. Springer LNCS 1870, 2000.

Parameterized Models for On-line and Off-line Use

Pieter Wuille, Tom Schrijvers*

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. The Monadic Constraint Programming framework leverages Haskell's rich static type system and powerful abstraction mechanisms to implement an embedded domain specific language (EDSL) for constraint programming.
In this paper we show how the same constraint model expressed in the EDSL can be processed in various modes by external constraint solvers. We distinguish between *on-line* and *off-line* use of solvers. In off-line mode, the model is not solved; instead it is compiled to lower-level code that will search for solutions when compiled and run. For on-line use, the search can be handled by either the framework or in the external solver. Off-line mode requires recompilation after each change to the model. To avoid repeated recompilation, we separate model from data by means of parameters that need not be known at compile time. Parametrization poses several challenges, which we resolve by embedding the EDSL more deeply.

1 Introduction

The Monadic Constraint Programming framework integrates constraint programming in the functional programming language Haskell [1] as a deeply embedded domain specific language (EDSL). This has a considerable advantage compared to special-purpose Functional Constraint (Logic) Programming (FCP) languages such as Curry [2] or TOY [3]. We directly obtain state-of-the-art functional programming support with zero effort, allowing us to focus on constraint programming itself.

While the integration is not as tight, Haskell does offer good EDSL support to make the embedding quite convenient. Moreover, being less tight does provide for greater flexibility. Aspects that are baked into some FCP languages, such as search strategies or the particular solver used, are much more easily interchanged from within the program. In addition, the deep embedding of the EDSL allows us to use the constraint model for more than straight (on-line) solving. For instance, transformations can be applied to the model for optimization purposes or to better target a particular constraint solver. Alternatively, the model does not have to be solved on-line, but can drive a code generator that produces an executable for off-line solving.

* Post-Doctoral Researcher of the Research Foundation– Flanders (FWO-Vlaanderen).

This paper reports on the FD-MCP module of the framework, specific to finite domain (FD) solvers. We show how the framework supports different modes of processing an FD model, by both on-line and off-line solvers. Then we identify the need for parametrized models to make the off-line solver approach both more useful and more efficient. We show how the framework is adjusted to support parametrized models, including deeply embedded iteration constructs and indexed collections of constraint variables.

2 Monadic Constraint Programming

The MCP [4] framework is a highly generic constraint programming framework for Haskell. It provides abstractions for writing constraint models, constraint solvers and search strategies. This paper focuses on the solving and modeling parts.

2.1 Generic Constraint Programming Infrastructure

MCP defines type classes, Haskell's form of interfaces, for `Solver`s and `Term`s:

```
class Monad s => Solver s where
  type Constraint s :: *
  type Label s :: *
  add :: Constraint s -> s Bool
  run :: s a -> a
  mark :: s (Label s)
  goto :: Label s -> s ()

class Solver s => Term s t where
  newvar :: s t
```

A type that implements the `Solver` type class must provide a type[1] to represent its constraints and labels, an `add` function for adding constraints, a `run` function to extract the results, a tt mark function to create a label of its current state, and a `goto` function to return to a previous state.

A solver type *s* must also be a monad [5]. A monadic value `s a` is an abstraction of a form of computation `s` that yields a result `a`. Constraint solvers are typically computations that thread an implicit state: the constraint store.

A solver also provides one or more types of terms: `Term s t` expresses that `t` is a term type of solver type `s`. Each term type provides a method `newvar` to generate new constraint variables of that type.

MCP also defines a data type `Model`, representing a model tree:

```
data Model s a
 = Return a                              -- return a value
 | Add (Constraint s) (Model s a)        -- add a constraint
```

[1] Implemented using associated types in Haskell

```
  | Try (Model s a) (Model s a)          -- disjunction
    ...
```

The model tree is parametrized in the constraint solver `s` and returned result type `a`. This provides a type-safe way for representing constraint problem models for arbitrary solvers and result types.

On top of the model data type, MCP provides syntactic sugar (functions that construct model trees), such as `exists` (create a variable), `exist n` (create a list of `n` variables), `addC` (add a constraint), `/\` (conjunction), `\/` (disjunction), `conj` (conjunction of list of models), ... Finally, `Model s` is also a monad.

2.2 The FD-MCP Module

The FD-MCP framework introduces an extra layer of abstraction between the more generic `Solver` interface of the MCP framework and the concrete solver implementations.

In contrast to MCP's generic `Solver` interface, which is parametric in the constraint domain, the `FDSolver` interface of FD-MCP is fully aware of the finite domain (FD) constraint domain: both its syntax (terms and constraints) and meaning (constraint theory). It does however make abstraction of the particular FD solver and e.g., propagation techniques used. Hence, it provides a uniform modeling language that abstracts from the syntactic differences between different FD solvers.

On the one hand, this allows the development of solver-independent models, model transformations (e.g., for optimization) and model abstractions (capturing frequently used patterns). On the other hand, specific solvers may focus on the efficient processing of their constraint primitives without worrying about modeling infrastructure.

FD-MCP Modeling Primitives The FD-MCP modeling language is built as a wrapper on top of the MCP solver interface. This way, the domain-independent combinators of the MCP framework, such as conjunction `(/\)` and existential quantification `exist` are available for FD models. The FD-MCP modeling language adds FD-specific constructs to that. Advanced FD constructs are defined in terms of a small set of core primitives, resulting in a layered structure.

The core constraints and terms are defined by the `FDConstraint` and `FDExpr` types respectively:

```
data FDConstraint s
 = Less (FDExpr s) (FDExpr s)   -- [[Less x y]]   ≡ [[x]] < [[y]]
 | Diff (FDExpr s) (FDExpr s)   -- [[Diff x y]]   ≡ [[x]] ≠ [[y]]
 | Same (FDExpr s) (FDExpr s)   -- [[Same x y]]   ≡ [[x]] = [[y]]
 | Dom (FDExpr s) Int Int       -- [[Dom x y z]]  ≡ [[x]] ∈ {y,...,z}
 | AllDiff [FDExpr s]           -- [[AllDiff [x1,...,xn]]] ≡ ⋀_{i≠j} xi ≠ xj
 | ...
```

```
data FDExpr s
 = Var (FDTerm s)                -- ⟦Var v⟧      ≡ ⟦v⟧
 | Const Int                     -- ⟦Const n⟧    ≡ n
 | Plus (FDExpr s) (FDExpr s)    -- ⟦Plus x y⟧   ≡ ⟦x⟧ + ⟦y⟧
 | Minus (FDExpr s) (FDExpr s)   -- ⟦Minus x y⟧  ≡ ⟦x⟧ - ⟦y⟧
 | Mult (FDExpr s) (FDExpr s)    -- ⟦Mult x y⟧   ≡ ⟦x⟧ * ⟦y⟧
 | ...
```

where the comment after each constructor shows its denotation, and `FDTerm s` refers to the type of the terms used to represent FD variables.

On top of the core primitives, a number of convenient abstractions and syntactic sugar exist. Firstly, standard arithmetic operators and integer literals can be used for `FDExpr s` thanks to an implementation of Haskell's `Num` type class. Thus `Plus x (Mult (Const 2) y)` can be written succinctly as `x + 2 * y`. More syntactic sugar exists for writing constraints:

```
x @<  y      = Add (Less x y) true
x @>  y      = y @< x
x @>= y      = x + 1 @> y
x @: (l,u)   = Dom x l u
xs `allin` d = conj [ x @: d | x <- xs ]
...
```

Mapping to the solver backend The backend takes care of compiling an FD-MCP model to a particular FD solver. To enable this compilation, the solver must implement the following `FDSolver` type class (in addition to implementing the `Solver` type class of the MCP framework):

```
class Term s (FDTerm s) => FDSolver s where
  type FDTerm s :: *
  compile_constraint :: FDConstraint s -> Model s Bool
```

The `FDSolver` type class makes two demands of a solver `s`:

- It must provide an (associated) type `FDTerm s` for its terms.
- The function `compile_constraint` must take care of converting from an individual FD-MCP constraint to a model for the solver. Note that, to allow mapping a single FD-MCP constraint to a conjunction of solver constraints involving auxiliary variables, this function returns a model rather than a single constraint. This model is not allowed to contain any disjunctions.

The following law specifies the `compile_constraint` function:

Definition 1 (Denotation Preservation). *The `compile_constraint` function preserves denotation iff*

$$[\![\cdot]\!] \circ \mathit{compile\_constraint} \equiv [\![\cdot]\!]$$

where $[\![\cdot]\!]$ maps a model or constraint onto its denotation, i.e., its logical meaning, and $\equiv$ denotes extensional function equality. Two denotations are equal iff they are logically equivalent.

Integration with MCP The `FDSolver` type class allows us to define a generic solver `FDWrapper s` that encapsulates the mapping from the generic model to the solver-specific model.

The `FDWrapper s` is an MCP `Solver` which uses `FDConstraints` as constraints and `FDExprs` as terms.[2]

```
newtype FDWrapper s a = FDWrapper { unwrapFD :: s a }

instance FDSolver s => Solver (FDWrapper s) where
  type Constraint (FDWrapper s) = FDConstraint s
  add c = FDWrapper $ untree $ compile_constraint c
  ...
  run = run . unwrapFD

instance Term (FDWrapper s) (FDTerm t) where
  newvar = FDWrapper $ newvar >>= \x -> return $ Var x
  ...

untree :: Solver s => Model s a -> s a
untree ... = ...
```

The `add` function first converts one `FDConstraint s` to a model tree for the underlying solver `s` with the `compile_constraint` function. Then, it turns this tree into a single (wrapped) monad action for the underlying solver `s` using `untree`. While `untree` is a generic function that works on any Model, instances of FDSolver provide their own `compile_constraint` to do the translation to their internal constraints. A similar approach is used for the other solver methods. The `newvar` method of the wrapper requests a new variable from the underlying solver, and returns it wrapped in a `Var` constructor. This is why the `FDConstraint` and `FDExpr` structures, as well as `FDWrapper` itself, are parametrized in the underlying MCP solver `s`. Finally, `run` unwraps the encapsulated monad action and runs it.

3 Solver Backends and Modes

The initial release of the MCP framework featured only one solver, a simple FD solver implemented in Haskell. However, rather than implement a solver in Haskell, it is much more attractive to interface external state-of-the-art solvers implemented in lower-level languages. That is why we have recently provided an interface to the Gecode FD solver in C++ [6]. In this work we expand considerably upon this initial interface and show how the same external solver can be interfaced in different modes.

[2] The instance requires `s` to belong to the `FDSolver` class, which requires a type `t` to belong to class Term s, which requires `s` to belong to class `Solver` itself.

3.1 On-line and Off-line Modes

Firstly, we distinguish between *on-line* and *off-line* use. The former means that the constraint model is processed by the MCP framework, in collaboration with the solver, to produce solutions. This mode is used for the original Haskell-based FD solver. The latter concerns *staged compilation*: in the first stage, the FD model is processed by the MCP framework that produces code for the second stage in the solver's programming language; the stage-2 code produces solutions. This mode was used in the original Gecode backend of [6]. The off-line mode comes with a compilation function $\langle\!\langle \cdot \rangle\!\rangle :: \texttt{Model OfflineSolver}\ a \rightarrow \texttt{C++}$ instead of the usual `run` function for solvers.

The off-line mode has a clear appeal for performance reasons: it avoids the interpretative overhead when solving the constraint model in the second stage. Of course, there is the compilation overhead of the first stage. We come back to this issue in the next section, where we considerably improve the usefulness of the off-line mode.

The on-line mode is very convenient for programming the search: all the high-level search features of the MCP framework are available. In contrast, our off-line Gecode solver provides a fixed search strategy. A considerable disadvantage of the on-line mode is the interpretative overhead of Haskell, which is confounded by the fact that the FD solver is implemented in Haskell itself.

New on-line Gecode solver In this paper we present a new on-line mode for the external Gecode solver. This combines the performance of Gecode with the high-level search features of the MCP framework. The solver type is defined as:

```
newtype OnlineSolver a
  = OnlineSolver { runOnline :: StateT GecodeState IO a }
```

The `OnlineSolver` is a monad composition of:

the `IO` monad: to access the Gecode library through the Haskell Foreign Function Interface (FFI), and

the `StateT GecodeState` monad transformer: to maintain the solver state:

```
data GecodeState = GecodeState { space :: Space
                               , cexpr :: Map FDExpr IntTerm }
```

which consists of a reference to the current Gecode space, and a map to translate FD expressions in the constraint model to constraint variables in the Gecode solver.

The `OnlineSolver` is recognized as an actual solver by the framework with the following instance:

```
instance Solver OnlineSolver where
   type Constraint OnlineSolver =  GecodeConstraint
   add c  =  addOnlineGecode c
   run m  =  unsafePerformIO $ do state <- newState
```

```
                                  evalStateT state (runOnline m)
type Label OnlineSolver      =  GecodeState
mark   =  get
goto s =  copyState s >>= put
```

The supported constraints of this solver are of type `GecodeConstraint`. The `addOnlineGecode` function adds a contraint to the current Gecode space, through the FFI. This involves constructing the constraint arguments, the FD expressions, in the Gecode solver. The `cexpr` helps out here, capturing earlier mappings of constraint variables and other FD expressions already have a representation in the Gecode solver. This results in dynamic common subexpression elimination. Running the solver means running the underlying `IO` monad and the state transformer, with appropriate initial state.

Finally, for disjunctive models and branches in the search tree, we use the copying technique in Gecode. Thus for the label of a solver state, we simply use the solver state, i.e. the Gecode space, itself. Whenever creating a branch starting form a given space, we install a copy of that space as the current space so as not to affect other branches.

Thanks to this relatively simple instance, we can now use the MCP infrastructure (e.g., a search queue, compositional search transformers and enumeration) for the on-line Gecode solver.

3.2 Programmed versus Fixed Search Modes

The new on-line Gecode backend of the framework offloads constraint propagation on the Gecode solver, but still allows the programmer to program and specify the search heuristics through the high-level interface. We call this approach the *programmed search mode*. It has clear advantages in terms of expressivity, but it does incur an interpretative penalty for search, which for many constraint problems has a considerable impact on the overall solving time.

In order to avoid the interpretative overhead for search, we provide a second mode of on-line use, the *fixed search mode*. Just like the off-line Gecode solver, this mode provides a fixed search strategy implemented in C++ for the on-line Gecode solver. In this mode, labelling the model does not produce a whole subtree that is affected by the framework's search heuristics. Instead, a single node is generated on the MCP side that corresponds to many nodes in the Gecode solver which are processed by a fixed search strategy.

4 Parameterized Models

Many FD models are naturally parameterized in a problem size and/or other instance-specific integer values. For instance, the n-queens problem is parameterized in the board size, the Golomb ruler problem is parameterized in the ruler size, ...

Such parameterization does not pose any problem for the on-line solvers. The parameterized model is simply written as a *model function* from one (or more)

integer value to an FD model. An `FDModel` is simply a `Model` for an `FDSolver`, that returns a list of solutions.

```
pmodel :: Int -> FDModel s
pmodel = \n -> ...
```

In order to solve the model, the model function is applied to the appropriate values, and the resulting model is handed to the on-line solver. No surprises.

For off-line solvers, we could follow the same technique. However, then we would obtain a non-parameterized off-line executable. Each time we would like to change the parameters, we would have to generate a new off-line executable! That is very costly in terms of compilation times, compared to the on-line solvers. The latter require only one invocation of the Haskell compiler for a parameterized model, while the former requires one invocation of the Haskell compiler and subsequently, for each instantiation of the parameters, an invocation of the C++ compiler. Moreover, the size of the off-line code is dependent on the problem size, because the framework fully flattens the model before generating code. Hence, the larger the problem size, the bigger the generated C++ code, and the longer the C++ compilation times. In summary, a new approach is necessary to make parameterized models practical for off-line solving.

The remainder of this section shows our approach for representing and compiling parameterized models. It has the two desirable properties: 1) a parameterized model requires only a single invocation of the C++ compiler, and 2) the generated code is independent of the parameter value.

4.1 Parameters

We still represent parameterized models by model functions, but the functions take FD expression terms rather than integers as arguments.

```
pmodel :: FDExpr s -> FDModel s
pmodel = \n -> ...
```

For brevity, we will omit the type parameters `s` in further signatures mentioning FD expressions and models.

We still retain the above functionality for off-line solvers, as integer values can be lifted to FD expressions using the `Const :: Integer -> FDExpr` constructor. Moreover, `FDExpr` is also an instance of the `Num` type class, so integer literals can be supplied directly as arguments: `pmodel 1425`.

Of more interest is of course the treatment of model functions for off-line solving. A model function is compiled by applying it to special `FDExpr` values that represent *deferred values* . These deferred values will not be known until the C++ stage. We denote a deferred value in the first stage as `` `$p$ ``, where p is the corresponding representation, a C++ `int` variable, in the second stage.

So using these deferred variables, we again obtain an `FDModel` that can be compiled much as before. Only the deferred values require special care. They are mapped to `int` instance variables of the generated C++ class that represents the Gecode constraint model. A new instance of the problem is created by instantiating an object of that class with the desired integer values for the parameters.

4.2 Indexed Constraint Variable Collections

Unfortunately, this is not the end of the story. Parameters of type `FDExpr` have fewer uses than values of type `Integer`. Indeed, the former can be used as arguments to constraints, but the latter can appear in many useful Haskell library functions as well as several functions of the MCP framework. Perhaps the most essential such function is `exist :: Integer -> ([Term] -> FDModel) -> FDModel`, which creates the specified number of constraint variables. In many parametric models, the number of constraint variables depends on the parameter value.

However, for the off-line solver, the integer value of the parameter is not available. Thus the actual creation of the list must be deferred from the on-line Haskell phase to the off-line phase. Moreover, we may wish to use a different data structure than a linked list in the off-line phase, such as an array in C++.

Hence, to allow writing models that can be used with both on-line and off-line solvers, we introduce a type `FDCollection s` indexed by the solver type `s`. For on-line solvers like `OvertonFD`, it is defined as an on-line Haskell list:

```
type instance FDCollection OvertonFD = [FDTerm OvertonFD]
```

For off-line solvers like `OfflineGecode`, a deferred collection $`c$ is used that only records an identifier c of the particular collection:

```
type instance FDCollection OfflineGecode = OfflineCollection Int
```

However, when writing a constraint model that is polymorphic in the solver type, `FDCollection s` acts as an abstract data type that only allows a limited number of operations, supported by both on-line and off-line solvers. The foremost of these operations are:

- `fdexist :: FDExpr -> (FDCollection -> FDModel) -> FDModel` creates a new collection of specified size, and acts as a generalization of `exist`. This function is implemented in terms of `exist` for on-line solvers, but creates a new deferred collection for off-line solvers. Note that the size of generated code for the latter is constant (a single array declaration) as opposed to linear like `exist`.
- `(!) :: FDCollection -> FDExpr -> FDExpr` returns an element at a given index in the collection. For on-line solvers it is implemented in terms of list indexation `(!!)`, but for off-line solvers a term denoting deferred indexation is returned. Then we have that $\langle\!\langle `c \mathbin{!} i \rangle\!\rangle = c[\langle\!\langle i \rangle\!\rangle]$.
- `collect :: [FDExpr] -> FDCollection` turns a list of variables into a collection.

Global constraints form another class of functions that involve collections. These have been modified to support collections instead of Haskell lists:

- `allDiff :: FDCollection -> FDModel` all variables in the given collection are mutually distinct.

- `sorted :: FDCollection -> FDModel` the given collection is sorted.
- `allin :: FDCollection -> (FDExpr,FDExpr) -> FDModel`
  all variables in the given collection have a value between the given lower and upper bounds.

4.3 Iteration

Often the above operations for collections are not expressive enough. Instead of imposing global constraints on a collection or indexing specific entries, many models process all elements of a collection one at a time. For this purpose an iteration construct is necessary.

Iteration Primitives We introduce in our framework the iteration primitive `foreach :: (FDExpr,FDExpr) -> (FDExpr -> FDModel) -> FDModel`, whose denotation is:

$$[\![\texttt{foreach}\ (l,u)\ f]\!] \equiv \bigwedge_{i=l}^{u} [\![f\ i]\!]$$

For instance, we write $\bigwedge_{i=1}^{n}(c_i > i)$ as:

```
foreach (1,n) $ \i -> (c ! i) @> i
```

For on-line solvers, `foreach` is expanded literally according to its semantics:

```
foreach (l,u) f  = conj [ f i | i <- [l..u] ]
```

However, for off-line solvers, we may not know the range of the loop, if it depends on a model parameter. Even if we do know the range, we may choose not to flatten the loop if the range is too large. In these cases, `foreach` is compiled to a C++ `for`-loop:

$$\langle\!\langle \texttt{foreach}\ (l,u)\ f \rangle\!\rangle = \texttt{for (int i = } \langle\!\langle l \rangle\!\rangle \texttt{; i =< } \langle\!\langle u \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f\ \texttt{`i} \rangle\!\rangle \texttt{ \}}$$

So the size of the generated code is independent of the size of the iteration range.

Because iteration over the whole range, rather than a subrange, of a collection occurs quite frequently, we introduce a second iteration construct `forall :: FDCollection -> (FDTerm -> FDModel) -> FDModel`, whose denotation is:

$$[\![\texttt{forall}\ c\ f]\!] \equiv \bigwedge_{v \in c} [\![f\ c]\!]$$

For instance, we write $\bigwedge_{v \in c}(v > i)$ as:

```
forall c $ \v -> v @> i
```

Again the on-line solvers' implementation of `forall` is a direct transliteration of the denotation:

```
forall c f = conj [f v | v <- c]
```

For the off-line solver, we can define `forall` in terms of `foreach`, if we provide access to a collection's size with `size :: FDCollection -> FDExpr`:

```
forall c f = foreach (1, size c) $ \i -> f (c ! i)
```

which means that we get the following C++ code:

$$\langle\!\langle \texttt{forall `}c\ f \rangle\!\rangle = \texttt{for (int i = 0; i < } \langle\!\langle \texttt{size `}c \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f \texttt{ `}c\texttt{[i]} \rangle\!\rangle \texttt{ \}}$$

Example The following program is a parameterized model of the n-queens problem:

```
nqueens n =                          --  define model function 'nqueens'
  fdexist n $ \q -> do               --  new collection 'q' of size n
    q `allin` (1,n)                  --  all variables in range [1..n]
    foreach (1,n) $ \i ->            --  for i in [1..n]
      foreach (1,i) $ \j -> do       --  for j in [1..i]
        q!i     @/= q!j              --  \
        q!i + i @/= q!j + j          --  | constraints
        q!i - i @/= q!j - j          --  /
    return q                         --  return result collection
```

Except for import statements and a main function that inputs the parameter value, calls the solver and outputs results, this is a fully working Haskell program.

Derived Iteration Constructs Finally, collections need a range of utility functions like those supported on Haskell lists:

- `fdmap :: (FDExpr -> FDExpr) -> FDCollection -> Model s FDCollection` transforms each element of a collection using a specified function, similar to the standard Haskell function `map`.
- `fdfold :: (FDExpr -> FDExpr -> Model s FDExpr) -> FDExpr -> FDCollection -> Model s FDExpr` folds a collection to a single expression, similar to the standard Haskell function `foldl`.
- `fdappend :: FDCollection -> FDCollection -> Model s FDCollection` concatenates two collections, similar to the standard Haskell operator `(++)`.

Currently, we implement such utility functions on top of `fdexist` and `foreach`. For example, `fdmap` is implemented as follows:

```
fdmap f c =
  fdexist (size c) $ \result -> do
    foreach (1,size c) $ \i ->
      result!i @= f (c!i)
    return result
```

While this generic implementation is easy to define and works for both on-line and off-line solvers, it does introduces `(size c)` superfluous constraint variables. We will see in the evaluation section that this leads to performance degradation. Hence, we will add these functions as additional primitives in the framework, so solvers can provide their own optimized implementations.

5 Evaluation

In order to evalute the two new extensions, the on-line Gecode solver support and the support for parameterized models, existing benchmarks for Gecode have been ported to FD-MCP. Tables 1 and 2 show the results. Lines of code (LoC) are measured using SLOCcount[3], the timings in seconds are average CPU times over multiple runs.[4]

5.1 Solving Results

The first table lists the absolute timings for the original C++ benchmark, and the runtimes of the MCP versions relative to original benchmark. The columns show respectively: 1) the name of the benchmark and the parameter value (if any), 2) the runtime (in seconds) for the Gecode benchmark in C++, and the relative runtimes of 3) the C++ code generated in off-line mode, 4) the parametrized C++ code in off-line mode, 5) the on-line Gecode solver in programmed search mode, 6) the on-line Gecode solver in fixed search mode, and 7) the on-line Haskell-only solver. All of columns 3)-7) are based on the same MCP model. The − entries denote out of space, while + entries denote a time out (no result after 5 minutes).

Any relative timings close to 100% indicate that the corresponding MCP mode is a valid alternative for direct Gecode implementation in terms of efficiency. Clearly, the pure Haskell solver cannot compete with a native Gecode implementation. Hence, it has been worthwhile to invest in Gecode backends for MCP.

A few times we observe that the compiled code generated in MCP off-line mode is slightly faster than the original Gecode benchmark. This is likely due to start-up overhead where the absolute runtime is only a few milliseconds.

Finally, the MCP versions of the *partition* and *magicsquare* benchmarks depend heavily on `fdmap`, `fdfold` and `fdappend`, which introduce auxiliary constraint variables responsible for the dramatic runtime increase. *magicsquare* introduces $4n^2 + 4n$ superfluous variables, *partition* $8n$. This suggests that optimized versions for these functions are essential to be competitive.

[3] `http://www.dwheeler.com/sloccount/`

[4] Benchmarks have been performed on a 64-bit Ubuntu 9.04 system using a 1.67GHz Intel® Core™2 Duo T5500 processor, with 1GiB RAM. Software versions: GHC 6.10.3, GNU G++ 4.3.3, Gecode 3.1.0.

Benchmark		Gecode (s)	MCP (%)				
			Off-line		On-line		
		C++	C++	param C++	Gecode -	Gecode +srch.	pure Haskell
allinterval	4	0.006s	85.4%	85.4%	113%	112%	122%
	8	0.006s	91.5%	92.0%	223%	125%	964%
	13	3.52s	108%	108%	462%	94.0%	5640%
	15	120s	107%	108%	-	94.7%	+
queens	5	0.006s	83.6%	84.0%	131%	127%	180%
	13	0.007s	89.3%	89.6%	249%	222%	1870%
	27	0.008s	80.0%	80.7%	65000%	806%	+
	100	0.057s	45.2%	44.9%	+	2400%	+
partition	4	0.006s	87.1%	87.3%	181%	169%	212%
	8	0.007s	118%	118%	294%	216%	995%
	16	0.047s	6500%	6500%	31000%	8200%	210000%
	20	0.15s	49000%	48000%	-	140000%	-
magicsquare	3	0.006s	87.8%	88.1%	211%	203%	267%
	4	0.019s	34.2%	34.2%	104%	91.6%	288%
	5	0.83s	8.9%	9.0%	64.5%	7.3%	1300%
	6	0.007s	64000%	64000%	-	35000%	+

Table 1. Timings

5.2 Compilation Results

The columns of Table 2 show the name and parameter value of benchmarks, and the number of lines of code and their compilation times for the original C++ benchmark, the benchmark implemented in Haskell using MCP, and the generated C++ code both with and without parameters (unfolded).

These results clearly suport two conclusions:

1. Models written in MCP are more concise than in Gecode, and
2. Parametrized generated code avoids parameter-dependant code sizes.

6 Related and Future Work

There is wide range of CP systems and languages. For lack of space we only mention a few. We classify them according to the distinction made in Section 1. A more extensive overview of related work can be found in [4].

Stand-alone modeling languages Zinc [7] is a stand-alone modeling language. Model transformations and compilation processes to different constraint solver backends are implemented in a second language, Cadmium, which is based on ACD term rewriting [8].

Rules2CP [9] is another stand-alone modeling language. The compilation of Rules2CP to SICStus Prolog is also specified by rewrite rules.

Benchmark		Lines of code				Compilation time(s)			
		C++	MCP			C++	MCP		
				C++	unfold. C++			C++	unfold. C++
allinterval	3	52	19	61	71	3.3	0.22	2.7	2.0
	15				179				2.7
queens	4	80	14	53	79	3.4	0.21	2.5	2.1
	100				534				>20
partition	4	74	34	103	103	3.3	0.21	3.1	2.2
	20				295				4.5
magicsquare	3	62	35	94	92	3.4	0.22	2.9	2.8
	6				206				3.1

Table 2. Lines of code and compilation times

Constraint Programming API's Two other functional programming languages which provide CP support are Alice ML[5] and FaCiLe [10]. While Alice ML provides a run-time interface to Gecode [11], FaCiLe uses its own constraint solver in OCaml. Both provide a rather low-level and imperative API, which corresponds to the C++ API of Gecode in the case of Alice ML, and relies on side effects. Neither supports alternative backends or model transformations.

Integrations Cipriano et al. [12] translate constraint models written in both Prolog CLP(FD) and (Mini)Zinc to Gecode via an intermediate language called CNT without loop constructs. The transformation from CNT to Gecode is implemented in Haskell. In order to avoid the Gecode code blow-up, it attempts to identify loops in the unrolled CNT model. It also performs a number of simplifications in the model. Our approach is much more convenient and efficient, providing explicit looping constructs and compiling these directly without intermediate loop unrolling, and with strong guarantees that loops remain loops.

Future work The benchmarks clearly indicate that additional iteration primitives must be added to the framework, in order to be competitive with Gecode. The support for collections should also be further extended to *multi-dimensional indexing*, which is quite convenient for modelling grid-based problems like sudoku, and *collection parameters* for providing a variable number of deferred data such as supply and demand quantities in a transportation problem.

7 Conclusions

We have shown how to link the FD-MCP framework with Gecode to allow efficient on-line solving of constraint problems modeled using it. Furthermore, we added deferred parameters and indexable collections to the provided abstractions, allowing shorter and more useful off-line code to be generated. These extensions were

[5] `http://www.ps.uni-sb.de/alice`

implemented[6] and benchmarks show that there is often only a small performance penalty compared to native C++ implementations.

Acknowledgments We are grateful to Peter Stuckey for his helpful comments.

References

1. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming **13**(1) (Jan 2003) 0–255
2. Hanus (ed.), M.: Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org` (2006)
3. Fernandez, A.J., Hortala-Gonzalez, T., Saenz-Perez, F., Del Vado-Virseda, R.: Constraint functional logic programming over finite domains. Theory Pract. Log. Program. **7**(5) (2007) 537–582
4. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic Constraint Programming. J. Func. Prog. **19**(6) (2009) 663–697
5. Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, London, UK (1995) 24–52
6. Wuille, P., Schrijvers, T.: Monadic Constraint Programming with Gecode. In: Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation. (2009) 171–185
7. Marriott, K., et al.: The design of the Zinc modelling language. Constraints **13**(3) (2008) 229–267
8. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In Etalle, S., Truszczynski, M., eds.: ICLP. Volume 4079 of LNCS. (2006) 117–131
9. Fages, F., Martin, J.: From Rules to Constraint Programs with the Rules2CP Modelling Language. In: Recent Advances in Constraints. LNAI (2009)
10. Barnier, N.: Application de la programmation par contraintes à des problèmes de gestion du trafic aérien. PhD thesis, Institut National Polytechnique de Toulouse (December 2002) `http://www.recherche.enac.fr/opti/papers/thesis/`.
11. Gecode Team: Gecode: Generic constraint development environment (2006) Available from `http://www.gecode.org`.
12. Cipriano, R., Dovier, A., Mauro, J.: Compiling and executing declarative modeling languages to Gecode. In de la Banda, M.G., Pontelli, E., eds.: ICLP. Volume 5366 of LNCS. (2008) 744–748

[6] Available at `http://www.cs.kuleuven.be/~toms/MCP/`

A Denotational Semantics for Curry
(progress report)

Jan Christiansen[1], Daniel Seidel[2*], Janis Voigtländer[2]

[1] Christian-Albrechts-Universität Kiel
jac@informatik.uni-kiel.de
[2] Rheinische Friedrich-Wilhelms-Universität Bonn
{ds,jv}@informatik.uni-bonn.de

Abstract. We aim to build a denotational semantics for the functional logic programming language Curry, to be used for parametricity and logical relation arguments. First, we investigate only a subset of Curry, but include the important features that separate Curry from a just functional language. We compare a poweralgebraic and a multialgebraic semantic approach and motivate our decision for the multialgebraic one. Afterwards, we describe how general recursion and lists, as an example for algebraic data types, can be added.

1 Introduction

In functional languages, reasoning about program transformations and code verification only dependent on type information is a common technique. An underlying theoretical foundation are parametricity and free theorems [1,2]. In [3] we present in an example-driven way how free theorems adapt to functional logic languages, in particular to Curry. But a formalization and generalization of these results is still missing.

The natural way to formally investigate free theorems is to define a logical relation over the type system of the programming language. To allow for such an investigation, a completely compositional semantics is desirable. From experience, we would prefer a denotational semantics. The nearly standard semantics for functional logic programming is CRWL [4]. CRWL is a constructor based rewrite logic. That is, it consists of a set of (conditional) rewrite rules by which a program is rewritten to a set of constructor terms. Originally, CRWL does not consider type restrictions, does not support higher-order functions, and is not compositional in the sense a denotational semantics is. Even if known extensions for each of these aspects are considered, the style of a denotational semantics seems more valuable for our purpose. Hence, the first step in a formal investigation into free theorems for Curry is to develop an appropriate denotational semantics for at least a subset of the language reflecting the main functional logic features.

A functional logic language can be seen as a functional language extended by non-determinism and free variables. In general, adding non-determinism can

* This author was supported by the DFG under grant VO 1512/1-1.

be modeled by switching from a single value semantics to a set value semantics for terms. To set up a denotational semantics, a number of choices have to be made. In fact, [5] presents twelve possible points in the design space that span over three independent issues. One can decide about strict/non-strict functions, call-time/run-time choice, and angelic/demonic/erratic view of non-deterministic choice. Curry implements non-strict functions and call-time choice, and conceptually aims for the angelic view. Call-time choice means that a non-deterministic choice is made only once when a non-deterministic value is shared. It is best explained by the *double coin* example, where *coin* is non-deterministically 0 or 1 and *double* adds a number to itself. Since the choice for *coin* is shared in the application of *double*, the result of *double coin* is either 0 or 2, but never 1.[3]

2 A Simple Subset of Curry

We start by investigating a calculus modeling only a subset of Curry. It is a polymorphically typed lambda calculus with added base type Int, addition "+" on it, a primitive "?" that models non-deterministic choice, and constants $\mathsf{unknown}_\tau$ and failed_τ modeling free variables and failure, respectively. The type and term syntax are given by

$$\begin{aligned}\tau &::= \alpha \mid \mathsf{Int} \mid \tau \to \tau \\ t &::= x \mid n \mid t + t \mid \lambda x :: \tau.t \mid t\ t \mid t\ ?\ t \mid \mathsf{unknown}_\tau \mid \mathsf{failed}_\tau\,,\end{aligned}$$

where α ranges over type variables, x over term variables, and n over the integers. The typing rules are as follows:

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash n :: \mathsf{Int} \qquad \Gamma \vdash \mathsf{failed}_\tau :: \tau \qquad \Gamma \vdash \mathsf{unknown}_\tau :: \tau$$

$$\frac{\Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (t_1\ ?\ t_2) :: \tau} \qquad \frac{\Gamma \vdash t_1 :: \mathsf{Int} \qquad \Gamma \vdash t_2 :: \mathsf{Int}}{\Gamma \vdash (t_1 + t_2) :: \mathsf{Int}}$$

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1.t) :: \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1\ t_2) :: \tau_2}$$

Standard conventions apply here. In particular, typing environments Γ are taken as sets of the form $\alpha_1, \ldots, \alpha_k, x_1 :: \tau_1, \ldots, x_l :: \tau_l$, where all free term variables in the right-hand side term of the typing judgment have to appear in Γ, and all type variables in the right-hand side term and type and in $\tau_1, \ldots, \tau_l$ have to appear among the $\alpha_1, \ldots, \alpha_k$.

The presented language is intended as a core language. Curry programs can be transformed into this core language (with the extensions from Section 4) by standard transformations. For example, the Curry function *fresh* :: $\mathsf{Int} \to \mathsf{Int}$ defined by *fresh* _ = x **where** x **free** is desugared to $\lambda y :: \mathsf{Int}.\mathsf{unknown}_{\mathsf{Int}}$. On a related note, identity of free variables can be expressed by appropriately using (or not using) several occurrences of $\mathsf{unknown}_\tau$. For example, if f is a

[3] A full calculation for this example using our semantics is given at the end of Section 3.

function of type $\tau \to \tau \to \tau$, then the difference between $f\ x\ y$ and $f\ x\ x$, with both x and y declared as **free**, is captured as $f\ \mathsf{unknown}_\tau\ \mathsf{unknown}_\tau$ vs. $(\lambda x :: \tau. f\ x\ x)\ \mathsf{unknown}_\tau$, where for the latter we could also have introduced the syntactic sugar **let** $x = \mathsf{unknown}_\tau$ **in** $f\ x\ x$.

3 Design of a Denotational Semantics

3.1 The Angelic View

As already mentioned, to model non-determinism, we interpret terms as *sets* and not as single values. The semantics of a term is the set of all its possible values. The view of non-deterministic choice determines how failure is handled. In the angelic view, an "angel" collects all non-deterministic choices that do not fail and simply ignores failure as long as there is at least one different result. Thus, for example the semantics of 3 and 3 ? $\mathsf{failed}_{\mathsf{Int}}$ cannot be distinguished.

The appropriate way to model that kind of non-determinism is the *Hoare powerdomain*. If we restrict ourselves to directed-complete partial orders (dcpos) as domains, an adaption of Theorem 6.2.13 from [6] allows us to define the Hoare powerdomain in terms of *Scott-closed subsets*. A subset A of a dcpo $(D, \sqsubseteq)$ is called Scott-closed if it is a lower set that is closed under the suprema of directed subsets. For a dcpo $(D, \sqsubseteq)$, a *lower set* A is a subset of D where $x \in A$ implies that all $y \in D$ with $y \sqsubseteq x$ are in A, too. A *directed subset* A is a non-empty subset where each two elements in A have a supremum in A.

Definition 1. *Let* $\mathbf{D} = (D, \sqsubseteq)$ *be a dcpo. The* Hoare powerdomain $\mathcal{P}_H(\mathbf{D}) = (P, \sqsubseteq^\uparrow)$ *of* $\mathbf{D}$ *is the complete lattice of all Scott-closed subsets of* D*. Union and intersection are interpreted as the actual set operations and* $\sqsubseteq^\uparrow$ *is the order induced by* $\cup$*, i.e., it is set inclusion.*

Infimum and supremum of $M \subseteq \mathcal{P}_H(\mathbf{D})$ *are defined by*

$$\bigcap M = \{x \in D \mid \forall m \in M.\ x \in m\}$$
$$\bigcup M = \bigcap \{m \in \mathcal{P}_H(\mathbf{D}) \mid \forall n \in M.\ n \sqsubseteq^\uparrow m\}$$

With the just given definition of the Hoare powerdomain we have fixed the domain structure to dcpos. Results from [6] guarantee that domain constructions preserve that structure.

An unsatisfying aspect of the above for a semantics to calculate with is the size of elements of the powerdomain. The use of lower sets means also the use of "big" sets because besides the maximal values a term evaluates to, all their partial approximations would be included as well. For example, the semantics of the identity function on the integers would not only contain the semantic function $id_{\mathbb{Z}}$, but also all partial identities, and, therefore, infinitely many elements. Hence, we are intending to define the Hoare powerdomain in an equivalent way, but with smaller elements. The basic idea is the following definition with m-subsets of a dcpo. If $(D, \sqsubseteq)$ is a dcpo, we say that a subset A of D is a *maxima subset (m-subset)* if for each element $x \in A$ there is no element $y \in A$ with $x \sqsubset y$.

Definition 2. *Let* $\mathbf{D} = (D, \sqsubseteq)$ *be a dcpo. Then* $\mathcal{P}^*_H(\mathbf{D}) = (P, \sqsubseteq^\uparrow)$ *is the complete lattice of all m-subsets of D. Intersection is interpreted by* $m \uplus n = \{x \in D \mid \exists y_1 \in m. \exists y_2 \in n.\ x \sqsubseteq y_1 \wedge x \sqsubseteq y_2\}\uparrow$ *and union by* $m \uplus n = (m \cup n)\uparrow$ *for every* $m, n \in \mathcal{P}^*_H(\mathbf{D})$*, where* $M\uparrow = \{x \in M \mid \forall y \in M.\ x \sqsubseteq y \Rightarrow x = y\}$*. Moreover,* $\sqsubseteq^\uparrow$ *is induced by* $\uplus$ *via* $n \sqsubseteq^\uparrow m$ *iff* $m \uplus n = m$*.*

Infimum and supremum of $M \subseteq \mathcal{P}^*_H(\mathbf{D})$ *are defined by*

$$\bigwedge M = \{x \in D \mid \forall m \in M. \exists y \in m.\ x \sqsubseteq y\}\uparrow$$
$$\bigvee M = \bigwedge \{m \in \mathcal{P}^*_H(\mathbf{D}) \mid \forall n \in M.\ n \sqsubseteq^\uparrow m\}$$

Note that choosing identical names for the orders in Definitions 1 and 2, namely $\sqsubseteq^\uparrow$, was not by accident. For each dcpo $\mathbf{D} = (D, \sqsubseteq)$ both can be characterized as restrictions of the *Hoare-lifting* $\sqsubseteq^\uparrow$, defined on the power set of D by $A \sqsubseteq^\uparrow B \Leftrightarrow \forall a \in A. \exists b \in B.\ a \sqsubseteq b$, to $\mathcal{P}_H(\mathbf{D})$ and $\mathcal{P}^*_H(\mathbf{D})$, respectively. The following claim sets up an order isomorphism between $\mathcal{P}_H(\mathbf{D})$ and $\mathcal{P}^*_H(\mathbf{D})$ for every dcpo $\mathbf{D}$.

Claim. Let $\mathbf{D} = (D, \sqsubseteq)$ be a dcpo. Then $\mathcal{P}_H(\mathbf{D})$ and $\mathcal{P}^*_H(\mathbf{D})$ are order isomorphic. The functions $(\cdot)\uparrow : \mathcal{P}_H(\mathbf{D}) \to \mathcal{P}^*_H(\mathbf{D})$ with $M\uparrow = \{x \in M \mid \forall y \in M.\ x \sqsubseteq y \Rightarrow x = y\}$ and $(\cdot)\downarrow : \mathcal{P}^*_H(\mathbf{D}) \to \mathcal{P}_H(\mathbf{D})$ with $M\downarrow = \{x \in D \mid \exists y \in M.\ x \sqsubseteq y\}$ form an order isomorphism.

The claim enables us to work with $\mathcal{P}^*_H(\cdot)$ instead of $\mathcal{P}_H(\cdot)$ as powerdomain constructor. Without the claim, the semantics defined below could easily be adapted to use $\mathcal{P}_H(\cdot)$, but actual calculating then becomes less practical.

3.2 The Function Model

A last choice to be made is the model of non-deterministic functions. In [7] an overview over a range of approaches is given. The essential criterion for choosing a suitable function model is the decision about call-time vs. run-time choice. As mentioned in the introduction, Curry implements call-time choice. That restricts the suitable function models to either the *poweralgebraic* model with the restriction to *additive* functions or the *multialgebraic* model.

The poweralgebraic model interprets functions as maps from *sets to sets*. It can originally capture both, run-time and call-time choice. The additivity restriction fixes it as a model for call-time choice. It says that for every (semantic) function f and input sets X, Y we have $f(X \uplus Y) = f(X) \uplus f(Y)$. This in particular means that (at least on finite sets) every function is completely determined by its behavior on the empty set and all singleton sets. To satisfy additivity when evaluating terms, we have to ensure that shared variables embody the same non-deterministic choice. Let us consider the semantics of $(\lambda x :: \mathsf{Int}.x + x)\ (0\ ?\ 1)$, which is *double coin* from the introduction. Clearly, the semantics of $0\ ?\ 1$ should be $\{0, 1\}$. But, maybe not so obviously, by additivity/call-time choice the semantics of the whole term should be only the sum of two zeros or two ones, and, therefore, the set $\{0, 2\}$. Thus, for a term semantics that guarantees additivity of functions we could either define function application by providing an input

set, to a function, piecewise as singleton sets (and maybe only pass through the empty set, with some extra care to guarantee laziness) or we could carefully have an eye on the function input whenever it is used more than once in the function body. The latter approach is preferable in an implementation as in [8]. But it is rather unwieldy in the design of a denotational semantics, because we would have to either deeply investigate the syntax of terms or suspect a necessity for sharing whenever evaluation of a term splits into the evaluation of several subterms. Thus, the first alternative mentioned above seems more adequate. Nevertheless, we abstain from providing a poweralgebraic semantics based on it. The reason is simple. It turns out that such a semantics would be nearly identical to the multialgebraic semantics we are going to provide, and which provides a better overall intuition about the actual behavior of functions, as explained next.

A function taking only singleton sets or the empty set can also be seen as a function taking single elements, where the empty set has to be modeled as a special element. Mapping *elements to sets* (instead of *sets to sets*) is exactly the multialgebraic approach. As a consequence, we consider the multialgebraic approach as more natural than the poweralgebraic approach in the context of call-time choice.

Type and term semantics for the multialgebraic approach are given in Fig. 1. Types are interpreted as the domains of individual semantic objects, not as the corresponding powerdomains. Therefore, the semantics of a term of type τ is in the powerdomain of τ's semantics. The type Int is interpreted as the set of integers with the discrete order, *without* a least element. The type semantics for a polymorphic type is fixed by a type environment θ that maps type variables to dcpos.

Regarding the function space, functions take exactly *one element* of the input type to *a set of elements* of the output type, i.e. are maps from a domain to a powerdomain. This is what makes the model multialgebraic [7]. Failure (essentially, the empty set) as input to a function needs extra handling, as already mentioned. It is not present in any type's domain, but can conceptually be a valid input to a function, hence we add it as a special element $\bot$ to the possible inputs of a function. This is realized by the lifting operation $(\cdot)_\bot$ that adds $\bot$ as least element to a dcpo. Continuity[4] of functions is enforced explicitly to guarantee that the function space itself is a dcpo. The order on the function space is defined point-wise. Furthermore, the *least defined function* $\Omega = \lambda a.\emptyset$ is excluded from the function space, since we identify it with failure, which is not represented in the domains and instead comes in as the empty set in the respective powerdomain.[5]

For the term semantics the term environment σ maps from term variables to single elements of a domain lifted by $(\cdot)_\bot$. This reflects that semantic functions take single elements as input. For function application, the function argument, which is a set, has to be provided to a function element by element. Special care is needed for the empty set as argument. In that case we instead feed the value

[4] Continuity is understood as Scott-continuity, i.e. monotonicity and preservation of suprema of directed sets.

[5] We cannot distinguish $\lambda\_ \to \mathsf{failed}$ and failed in the absence of strict evaluation.

type semantics

$$
\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![\mathsf{Int}]\!]_\theta &= \mathbb{Z} \\
[\![\tau_1 \to \tau_2]\!]_\theta &= \{f : ([\![\tau_1]\!]_\theta)_\perp \to \mathcal{P}^*_H([\![\tau_2]\!]_\theta) \mid f \text{ continuous}\} \setminus \{\Omega\}
\end{aligned}
$$

term semantics

$$
\begin{aligned}
[\![x]\!]_{\theta,\sigma} &= \begin{cases} \emptyset & \text{if } \sigma(x) = \perp \\ \{\sigma(x)\} & \text{otherwise} \end{cases} \\
[\![n]\!]_{\theta,\sigma} &= \{n\} \qquad \forall n \in \mathbb{Z} \\
[\![t_1 + t_2]\!]_{\theta,\sigma} &= \bigcup_{a \in [\![t_1]\!]_{\theta,\sigma}} \bigcup_{b \in [\![t_2]\!]_{\theta,\sigma}} \{a + b\} \\
[\![\lambda x :: \tau.t]\!]_{\theta,\sigma} &= \{\lambda a.[\![t]\!]_{\theta,\sigma[x \mapsto a]}\} \setminus \{\Omega\} \\
[\![t_1\ t_2]\!]_{\theta,\sigma} &= \bigcup_{f \in [\![t_1]\!]_{\theta,\sigma}} \biguplus_{a \in [\![t_2]\!]_{\theta,\sigma}} (f\ a) \\
[\![t_1\ ?\ t_2]\!]_{\theta,\sigma} &= [\![t_1]\!]_{\theta,\sigma} \uplus [\![t_2]\!]_{\theta,\sigma} \\
[\![\mathsf{unknown}_\tau]\!]_{\theta,\sigma} &= ([\![\tau]\!]_\theta)\uparrow \\
[\![\mathsf{failed}_\tau]\!]_{\theta,\sigma} &= \emptyset
\end{aligned}
$$

Fig. 1. Type and Term Semantics for the Multialgebraic Model

$\perp$ to the function under consideration. This is reflected in the definition of the lazy union operator:

$$
\biguplus_{a \in A} = \begin{cases} \bigcup_{a \in \{\perp\}} & \text{if } A = \emptyset \\ \bigcup_{a \in A} & \text{otherwise}\,. \end{cases}
$$

We can state type correctness of the semantics as follows.

Lemma 1. *Let $\Gamma \vdash t :: \tau$. Then for every θ that maps each type variable in Γ to a dcpo and every σ such that for each term variable $x :: \tau'$ in Γ, $\sigma(x) \in ([\![\tau']\!]_\theta)_\perp$, we have $[\![t]\!]_{\theta,\sigma} \in \mathcal{P}^*_H([\![\tau]\!]_\theta)$.*

We conclude the section by a formal calculation of the semantics of $(\lambda\ x :: \mathsf{Int}.x\ +\ x)\ (0\ ?\ 1)$. Since our semantics is completely compositional, we can precompute the semantics of the function and of the argument before we calculate the semantics of the whole term. We have:

$$
\begin{aligned}
[\![\lambda x :: \mathsf{Int}.x + x]\!]_{\emptyset,\emptyset} &= \{\lambda a.[\![x + x]\!]_{\emptyset,[x \to a]}\} \setminus \{\Omega\} \\
&= \{\lambda a. \bigcup_{b \in [\![x]\!]_{\emptyset,[x \to a]}} \bigcup_{c \in [\![x]\!]_{\emptyset,[x \to a]}} \{b + c\}\} \setminus \{\Omega\} \\
&= \{\lambda a. \bigcup_{b \in \{a\}} \bigcup_{c \in \{a\}} \{b + c\}\} \setminus \{\Omega\} = \{\lambda a.\{a + a\}\}
\end{aligned}
$$

And:

$$
[\![0\ ?\ 1]\!]_{\emptyset,\emptyset} = [\![0]\!]_{\emptyset,\emptyset} \uplus [\![1]\!]_{\emptyset,\emptyset} = \{0\} \uplus \{1\} = \{0, 1\}
$$

And finally:

$$\begin{aligned}
[\![(\lambda x :: \mathsf{Int}.x + x)\ (0\ ?\ 1)]\!]_{\emptyset,\emptyset} &= \bigcup_{f \in [\![\lambda x :: \mathsf{Int}.x+x]\!]_{\emptyset,\emptyset}} \biguplus_{a \in [\![0?1]\!]_{\emptyset,\emptyset}} (f\ a) \\
&= \bigcup_{f \in \{\lambda a.\{a+a\}\}} \biguplus_{a \in \{0,1\}} (f\ a) \\
&= ((\lambda a.\{a + a\})\ 0) \uplus ((\lambda a.\{a + a\})\ 1) \\
&= \{0 + 0\} \uplus \{1 + 1\} = \{0, 2\}
\end{aligned}$$

As was our intention, we get only 0 and 2 as results.

4 General Recursion and Algebraic Data Types

The subset of Curry that we handled so far includes the logical features of non-determinism and free variables, but not the functional features of general recursion and algebraic data types.

To add general recursion, we extend the term syntax by a primitive fix and add an appropriate typing rule. The semantics of $\mathsf{fix}\ t$ is, as usual, the least fixpoint of the semantics of (the function) t. The definition is shown in the last row of Fig. 2. That the definition characterizes the least fixpoint is a standard theorem for dcpos with least element (see [6]). Lemma 1 remains valid for the extension. Hence, general recursion integrates smoothly.

type semantics
$[\![[\tau]]\!]_\theta = lfp(\lambda S.\{[\,]\} \cup \{a : b \mid a \in ([\![\tau]\!]_\theta)_\perp, b \in S_\perp\})$

term semantics
$[\![[\,]_\tau]\!]_{\theta,\sigma} = \{[\,]\}$
$[\![t_1 : t_2]\!]_{\theta,\sigma} = \biguplus_{h \in [\![t_1]\!]_{\theta,\sigma}} \biguplus_{t \in [\![t_2]\!]_{\theta,\sigma}} \{h : t\}$
$[\![\mathsf{case}\ t\ \mathsf{of}\ \{[\,] \to t_1;\ x : xs \to t_2\}]\!]_{\theta,\sigma} = \bigcup_{l \in [\![t]\!]_{\theta,\sigma}} \begin{cases} [\![t_1]\!]_{\theta,\sigma} & \text{if } l = [\,] \\ [\![t_2]\!]_{\theta,\sigma[x \mapsto h, xs \mapsto t]} & \text{if } l = h : t \end{cases}$
$[\![\mathsf{fix}\ t]\!]_{\theta,\sigma} = \bigsqcup_{n \geqslant 0} (\lambda A. \bigcup_{f \in [\![t]\!]_{\theta,\sigma}} \biguplus_{a \in A} (f\ a))^n\ \emptyset$

Fig. 2. Extensions of Type and Term Semantics

To show how algebraic data types can be added to the semantics, we investigate lists. Therefore, we extend the type syntax by a list type $[\tau]$ and the term syntax by the list constructors $[\,]_\tau$ and $(:)$, as well as a case-statement for list destruction. Furthermore, we add the appropriate typing rules (not shown here).

Semantically, all non-determinism in lists can be flattened out, in the sense that there is only a top-level choice between different list structures, each of

which is completely deterministic (including having completely deterministic elements). This perspective is similar to the perspective on functions, and indeed we can regard the constructors $[\,]_\tau$ and (:) as a nullary and a binary function, respectively. This also explains that the arguments to (:) can be $\bot$. Concrete semantic definitions are given in Fig. 2. The dcpo order on the interpretation of list types is by element-wise comparison. Again, Lemma 1 remains valid for the extension.

In a call-time choice setting with set semantics it does not matter where non-determinism is "hidden" in an algebraic data type. For example, the terms [1 ? 2] and [1] ? [2] are to be considered semantically equivalent. That is why we "flatten out" all non-determinism. This way we guarantee that we have only *one* semantic representation for each conceptual value of list type. We calculate the semantics of [1 ? 2] and [1] ? [2] to show that they are indeed the same.

$$\begin{aligned}
[\![(1 \,?\, 2) : [\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset} &= \biguplus_{h \in [\![1?2]\!]_{\emptyset,\emptyset}} \biguplus_{t \in [\![[\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset}} \{h : t\} \\
&= \biguplus_{h \in ([\![1]\!]_{\emptyset,\emptyset} \uplus [\![2]\!]_{\emptyset,\emptyset})} \biguplus_{t \in [\![[\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset}} \{h : t\} \\
&= \biguplus_{h \in \{1,2\}} \biguplus_{t \in \{[\,]\}} \{h : t\} = \{1 : [\,], 2 : [\,]\} \\
[\![(1 : [\,]_{\mathsf{Int}}) \,?\, (2 : [\,]_{\mathsf{Int}})]\!]_{\emptyset,\emptyset} &= [\![1 : [\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset} \uplus [\![2 : [\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset} \\
&= (\biguplus_{h \in [\![1]\!]_{\emptyset,\emptyset}} \biguplus_{t \in [\![[\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset}} \{h : t\}) \\
&\quad \uplus (\biguplus_{h \in [\![2]\!]_{\emptyset,\emptyset}} \biguplus_{t \in [\![[\,]_{\mathsf{Int}}]\!]_{\emptyset,\emptyset}} \{h : t\}) \\
&= (\biguplus_{h \in \{1\}} \biguplus_{t \in \{[\,]\}} \{h : t\}) \uplus (\biguplus_{h \in \{2\}} \biguplus_{t \in \{[\,]\}} \{h : t\}) \\
&= \{1 : [\,]\} \uplus \{2 : [\,]\} = \{1 : [\,], 2 : [\,]\}
\end{aligned}$$

In contrast, in a poweralgebraic model the natural intuition would seem to be to see non-empty lists as non-deterministic heads followed by a non-deterministic choice of tails. In particular, of the latter some could be non-empty lists themselves, others empty lists or failures. This kind of nested non-determinism essentially prevents a unique representation for each conceptual value of list type. In particular, a seemingly straightforward poweralgebraic semantics for lists would falsely attribute more "definedness" to [1 ? 2] than to [1] ? [2].

The extended calculus is comprehensive enough to consider the examples from [3]. Calculations for a specific example are given in the next section.

5 An Example from [3]

We consider Example 3 from [3]. There, we found that in Curry free variables distort the standard (Haskell) free theorem for the type $[\alpha] \to [\alpha]$ in a particular way. While our measure of correctness there was running the Curry interpreter, we can now instead formally calculate using our semantics.

Translated into the syntax of our calculus (which is extended, along with the semantics, in the obvious way to deal with Booleans), we need the following

function definitions (where we have specialized *map* to a monomorphic type):

$$
\begin{aligned}
map &= \lambda h :: \mathsf{Int} \to \mathsf{Bool}.\mathsf{fix}\ (\lambda rec :: [\mathsf{Int}] \to [\mathsf{Bool}].\lambda l :: [\mathsf{Int}].\\
&\qquad\qquad \mathsf{case}\ l\ \mathsf{of}\ \{[\,] \to [\,]_{\mathsf{Bool}};\ x : xs \to (h\ x) : (rec\ xs)\})\\
f &= \lambda xs :: [\alpha].[\mathsf{unknown}_\alpha]\\
h &= \lambda x :: \mathsf{Int}.\mathsf{True}
\end{aligned}
$$

After some unfolding and massage, their semantics is obtained as follows:

$$
\begin{aligned}
[\![map]\!]_{\emptyset,\emptyset} &= \{\lambda h.\{\bigsqcup_{n\geqslant 0}(\lambda rec.\lambda l.\\
&\qquad\qquad \begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{y\in(h\ x)}\biguplus_{ys\in(rec\ xs)}\{y : ys\} & \text{if } l = x : xs\end{cases}{}^{n}\ \Omega\}\}\\
[\![f]\!]_{[\alpha\to\mathbf{D}],\emptyset} &= \{\lambda xs.\biguplus_{d\in(\mathbf{D}\uparrow)}\{[d]\}\}\\
[\![h]\!]_{\emptyset,\emptyset} &= \{\lambda x.\{\mathsf{True}\}\}
\end{aligned}
$$

Now we can calculate the semantics for parts of the expressions in Example 3 from [3]. For the sake of brevity, we omit type and term environments unless they are necessary for the calculation.

We start with the semantics of *map h*:

$$
\begin{aligned}
&[\![map\ h]\!]\\
&= \bigsqcup_{k\in[\![map]\!]}\biguplus_{l\in[\![h]\!]}(k\ l)\\
&= (\lambda h.\{\bigsqcup_{n\geqslant 0}(\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{y\in(h\ x)}\biguplus_{ys\in(rec\ xs)}\{y : ys\} & \text{if } l = x : xs\end{cases}{}^{n}\ \Omega\})\\
&\quad(\lambda x.\{\mathsf{True}\})\\
&= \{\bigsqcup_{n\geqslant 0}(\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{y\in((\lambda x.\{\mathsf{True}\})\ x)}\biguplus_{ys\in(rec\ xs)}\{y : ys\} & \text{if } l = x : xs\end{cases}{}^{n}\ \Omega\}\\
&= \{\bigsqcup_{n\geqslant 0}(\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{ys\in(rec\ xs)}\{\mathsf{True} : ys\} & \text{if } l = x : xs\end{cases}{}^{n}\ \Omega\}
\end{aligned}
$$

$$= \{(\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{ys\in(rec\ xs)}\{\mathsf{True} : ys\} & \text{if } l = x : xs\end{cases})$$
$$((\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{ys\in(rec\ xs)}\{\mathsf{True} : ys\} & \text{if } l = x : xs\end{cases})$$
$$\left(\bigsqcup_{n\geqslant 0}(\lambda rec.\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{ys\in(rec\ xs)}\{\mathsf{True} : ys\} & \text{if } l = x : xs\end{cases})^n\ \Omega\right))\}$$
$$= \{(\lambda l.\begin{cases}\{[\,]\} & \text{if } l = [\,]\\ \biguplus_{ys\in\begin{cases}\{[\,]\} & \text{if } xs = [\,]\\ \biguplus_{ys\in((\cdots)\ xs')}\{\mathsf{True} : ys\} & \text{if } xs = x' : xs'\end{cases}}\{\mathsf{True} : ys\} & \text{if } l = x : xs\end{cases})\}$$

Using this partial unrolling, we can calculate as follows:

$$\begin{aligned}[\![map\ h\ [0]]\!] &= \bigcup_{m\in[\![map\ h]\!]}\biguplus_{l\in[\![[0]]\!]}(m\ l)\\ &= \bigcup_{m\in[\![map\ h]\!]}(m\ [0]) = \{[\mathsf{True}]\}\end{aligned}$$

and thus:

$$\begin{aligned}[\![f\ (map\ h\ [0])]\!]_{[\alpha\to\{\mathsf{True},\mathsf{False}\}],\emptyset} &= \bigcup_{a\in[\![f]\!]_{[\alpha\to\{\mathsf{True},\mathsf{False}\}],\emptyset}}\biguplus_{b\in[\![map\ h\ [0]]\!]}(a\ b)\\ &= (\lambda xs.\{[\mathsf{True}],[\mathsf{False}]\})\ [\mathsf{True}] = \{[\mathsf{True}],[\mathsf{False}]\}\end{aligned}$$

On the other hand:

$$\begin{aligned}[\![f\ [0]]\!]_{[\alpha\to\mathbb{Z}],\emptyset} &= \bigcup_{a\in[\![f]\!]_{[\alpha\to\mathbb{Z}],\emptyset}}\biguplus_{b\in[\![[0]]\!]}(a\ b)\\ &= (\lambda xs.\{[z]\mid z\in\mathbb{Z}\})\ [0] = \{[z]\mid z\in\mathbb{Z}\}\end{aligned}$$

and thus:

$$\begin{aligned}[\![map\ h\ (f\ [0])]\!]_{[\alpha\to\mathbb{Z}],\emptyset} &= \bigcup_{c\in[\![map\ h]\!]}\biguplus_{d\in[\![f\ [0]]\!]_{[\alpha\to\mathbb{Z}],\emptyset}}(c\ d)\\ &= \bigcup_{z\in\mathbb{Z}}\bigcup_{c\in[\![map\ h]\!]}(c\ [z])\\ &= \bigcup_{z\in\mathbb{Z}}\{[\mathsf{True}]\} = \{[\mathsf{True}]\}\end{aligned}$$

This makes evident the failure of the standard free theorem, as observed in Example 3 from [3].

6 Conclusion

As we model most interesting functional logic features, we think that we have now a good base to formally investigate free theorems in a lazy functional logic

language, moving beyond examples as in [3] and the previous section. We also think that the denotational semantics will be of help for reasoning about functional logic programs in general.

We are aware that there is a wealth of existing work on the semantics of functional logic languages [9,4,10,11,12,13], but have not been able to make extensive comparisons here. In order to establish connections, we intend to prove our semantics equivalent to CRWL (or a corresponding extension of it).

Acknowledgment. We would like to thank Rudolf Berghammer and Achim Jung for discussions about (power)domain theoretic issues.

References

1. Reynolds, J.: Types, abstraction and parametric polymorphism. In: Information Processing, Proceedings, Elsevier (1983) 513–523
2. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture, Proceedings, ACM Press (1989) 347–359
3. Christiansen, J., Seidel, D., Voigtländer, J.: Free theorems for functional logic programs. In: Programming Languages meets Program Verification, Proceedings, ACM Press (2010)
4. González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming **40**(1) (1999) 47–87
5. Søndergaard, H., Sestoft, P.: Non-determinism in functional languages. The Computer Journal **35**(5) (1992) 514–523
6. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science. Oxford University Press (1994) 1–168
7. Walicki, M., Meldal, S.: Algebraic approaches to nondeterminism: An overview. ACM Computing Surveys **29**(1) (1997) 30–81
8. Braßel, B., Fischer, S., Hanus, M., Reck, F.: Transforming functional logic programs into monadic functional programs. In: Workshop on Functional and (Constraint) Logic Programming, Draft Proceedings. (2010)
9. Hanus, M., Lucas, S.: A denotational semantics for needed narrowing. In: Joint Conference on Declarative Programming, APPIA-GULP-PRODE'96, Proceedings. (1996) 259–270
10. Molina-Bravo, J., Pimentel, E.: Composing programs in a rewriting logic for declarative programming. Journal of Theory and Practice of Logic Programming **3**(2) (2003) 189–221
11. Tolmach, A., Antoy, S.: A monadic semantics for core Curry. In: Workshop on Functional and (Constraint) Logic Programming, Proceedings. Volume 86(3) of ENTCS., Elsevier (2003)
12. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Equivalence of two formal semantics for functional logic programs. In: Spanish Conference on Programming and Languages, PROLE'06, Proceedings. Volume 188 of ENTCS., Elsevier (2007) 117–142
13. Braßel, B., Berghammer, R.: Functional (logic) programs as equations over order-sorted algebras. In: Logic-Based Program Synthesis and Transformation, Pre-Proceedings. (2009)

A Declarative Debugger of Missing Answers for Functional and Logic Programming

On-going work on the system description ⋆

Fernando Pérez Morente, Rafael del Vado Vírseda

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
fperezmo@fdi.ucm.es rdelvado@sip.ucm.es

Abstract. *Declarative debugging* has many advantages over conventional approaches to debugging for functional and (constraint) logic programs. This paper presents a graphical declarative debugger of *missing computed answers* for the constraint lazy functional-logic programming system $\mathcal{TOY}$. Our debugger starts whenever a user finds that the set of computed answers for a given goal with finite search space misses some expected solution. Then, the system proceeds by exploring a *computation tree* that provides a declarative view of the *answer-collection* process performed by the computation, and it ends up with the detection of some function definition in the program that is incomplete. We provide the implementation of our debugging system, which analyzes a declarative view of the trace of the computation in order to simplify the process of reconstructing the computation tree. We have shown experimentally on a wide set of examples integrated in the $\mathcal{TOY}$ system that we are able to find some common errors in functional and logic programs.

1 Introduction

Debugging is one of the essentials parts of the software development cycle and a practical need for helping programmers to understand why their programs do not work as intended. For the purposes of this paper, debugging can be described as diagnosing the causes of unexpected results observed after the completion of some computation. Declarative (functional and logic) programming languages are intentionally designed to separate the logic of programs (*what* should be computed) from the operational behavior (*how* it should be computed), where features like *higher-order*, *polymorphism* or *lazy evaluation* make the computation more difficult to observe and understand. For this reason, the design of usable debugging tools becomes a difficult task because the traditional debugging techniques used in imperative languages, such as the *step-by-step* execution of the program, are not enough in declarative languages and users

⋆ This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407, S2009/TIC-1465, and UCM-BSCH-GR58/08-910502.

can hardly follow computation traces mainly due to lazy evaluation. Following a seminal idea from Shapiro [1], *declarative diagnosis* (a.k.a. *declarative debugging* or *algorithmic debugging*) proposes to replace computation traces by *Computation Trees* (briefly *CT*s) with results attached to their nodes, such that the result at any internal node follows from the results at the child nodes, using a program fragment also attached to the node. A *CT* whose root exhibits an unexpected result must include at least one so-called *buggy node*, whose result is unexpected but whose children's results are all expected. Buggy nodes can be located by navigating the *CT* with the help of an external *oracle* (usually the user) and point to program fragments responsible for the unexpected behavior.

Significant research on declarative diagnosis exists both for (lazy) functional programming (*FP*) and (constraint) logic programming (*CLP*) languages (see e.g., [2,3,4,5,6]). Unexpected results in *FP* are mainly *incorrect values*, while in *CLP* an unexpected result can be either a single computed answer regarded as *incorrect*, or a set of computed answers (for one and the same goal with a finite search space) regarded as *incomplete*. These two possibilities give rise to the declarative diagnosis of *wrong* and *missing* computed answers in *CLP*, respectively. In this paper we present the ongoing work describing a graphical declarative debugger of missing answers for the constraint lazy functional-logic programming system $\mathcal{TOY}$ [7]. The debugger implements the theoretical results presented in [8] for the declarative diagnosis of missing computed answers of lazy *FP* and *LP* languages, although the same principles can be applied to *CLP* languages. The system is available at `http://gpd.sip.ucm.es/rafav/`.

The rest of this paper is organized as follows: Section 2 presents informally the computation trees used in the rest of the paper. Section 3 shows a debugging session, which also introduces some of the features of the graphical interface. Section 4 provides some details about the implementation of the diagnosis method employed by the tool. The paper ends in Section 5 with some conclusions and some proposals for future work. For the convenience of reviewers, more additional examples and technical details can be found in the integrated version of the $\mathcal{TOY}$ system for missing answers provided in the Web page.

2 Declarative Debugging of Missing Answers

While methods and tools for the declarative diagnosis of *wrong answers* are known for constraint functional-logic programming languages [9,10,11,12,6], we are not aware of any practical tool or system in *CFLP* concerning the declarative diagnosis of *missing answers*, except our previous theoretical work [8]. However, missing answers, obtained when the set of computed answers for a given goal does not cover some expected answer, are a common problem which can arise even in the absence of wrong answers. A particular and important case of missing answers is manifested by a *finitely failing* goal for which solutions were expected. We provide a debugger of missing answers for the *CFLP* system $\mathcal{TOY}$ [7].

A $\mathcal{TOY}$ *program* $\mathcal{P}$ can include declarations of data, function types, type alias, infix operator declarations, and defining rules for function symbols. Two important syntactic categories in this setting are *expressions* and *patterns*. The

possible forms of an *expression* are $e ::= \bot \mid X \mid h \mid (e\, e')$, where X is a variable, h is either a function symbol or a data constructor, and $\bot$ is a symbol representing an *undefined value*. The symbol $\bot$ cannot be written directly in the programs, but it is important for representing correctly the *lazy semantics* of the language, and will be used by the debugger to indicate that a function call was not demanded during the analyzed computation. The expression $(e\, e')$ stands for the application of expression e to expression e'. Similarly, the possible forms of a *pattern* are $t ::= \bot \mid X \mid c\, \overline{t_n} \mid f\, \overline{t_m}$, where c represents a data constructor of arity greater or equal to n, and f a function symbol of arity greater than m. The defining rules for a function f have the form $f\, \overline{t_n} = e \Leftarrow C$, where the t_i are patterns, e is an expression, and C is a conjunction of *strict equalities* $e_1 == e_2$, evaluated to *true* if e_1 and e_2 can be reduced to some common *total* (i.e., without any occurrence of $\bot$) pattern. A *goal* G has the same form as the conditional part C. The computed *answer* for a goal G must be a substitution σ from variables to expressions (in particular patterns to represent values).

A declarative debugger for algorithmic diagnosis should look for a buggy node in a suitable CT in order to detect some incomplete function definition to be blamed for the missing answers. The approach in this paper uses CTs whose nodes have attached so-called *answer collection assertions*, briefly *aca*s. The *aca* at the root node has the form $G \Rightarrow \bigvee_{i \in I} \sigma_i$, asserting that all the solutions of the initial goal G are covered by the finite disjunction of computed answers $\bigvee_{i \in I} \sigma_i$ (*empty* in the case of a finitely failing goal). The *aca*s at internal nodes have the form $f\, \overline{e}_n \Rightarrow \bigvee_{i \in I} \sigma_i$, asserting that $\bigvee_{i \in I} \sigma_i$ covers all the solutions for an intermediate step $f\, \overline{e}_n$ intended to compute results of the *function call* $f\, \overline{e}_n$. In such function calls, the parameters $\overline{e}_n$ don't have to be totally evaluated; instead, they will be evaluated to the extent needed to solve the topmost expression under a *lazy evaluation strategy*. Moreover, the whole CT must be such that the validity of the *aca* at each node follows from the validity of the *aca*s at their children, under the assumption that the function definition relating the parent node to the children nodes is *complete* w.r.t. the intended program semantics. We satisfy this requirement building the CT as an *abbreviated proof tree* from a logically sound *inference system* for deriving *aca*s given in [8].

Once a CT has been constructed, the search for a buggy node can be implemented with the help of an external *oracle* (usually the programmer) who has a reliable declarative knowledge of the valid *aca*s, based on a so-called *intended interpretation* of the program. Any CT with an invalid *aca* at the root has surely at least one *buggy node* labeled with an invalid *aca* whose children are all labeled with valid *aca*s. Each buggy node N is related to some particular function f_N whose program rules are responsible for the computation of the *aca* at N from the *aca*s at N's children. Therefore, the program rules for f_N can be diagnosed as *incomplete*. The detection of buggy nodes depends on the oracle's judgements on the validity of *aca*s w.r.t. the intended meaning of the program. Since the oracle is usually the programmer, we can even experiment with different choices of the intended interpretation in order to obtain different diagnosis of possibly incomplete functions.

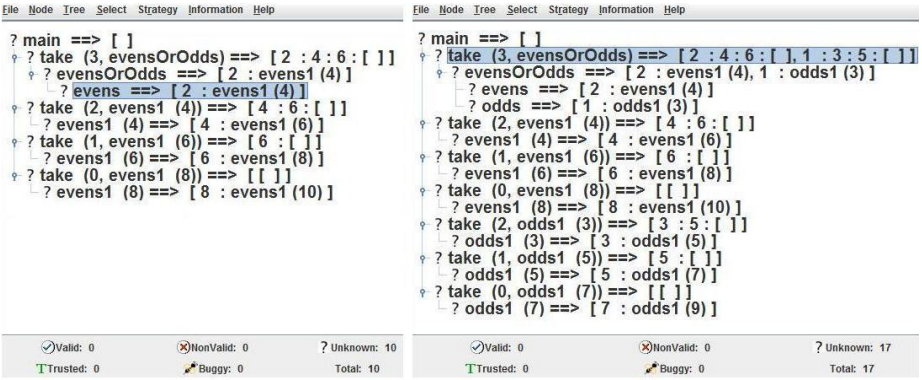

Fig. 1. *CT* displayed by the system, with missing answers (left) and complete (right).

3 A Debugging Session of Missing Answers in $\mathcal{TOY}$

In this section we show how our declarative debugger of missing answers works by means of a concrete example of its execution. The system can be downloaded from `http://gpd.sip.ucm.es/rafav/`. Assuming a correct implementation of the $\mathcal{TOY}$ system [7], if the computation of answers for a goal finishes after having collected finitely many answers, the user may decide that there are missing answers and type the command `/missing` at the system prompt in order to initiate a *debugging session*. The declarative debugger will graphically show the *CT* to help the user to find any function that is not computing all its values. We illustrate this process by means of the following program:

```
take :: int -> [A] -> [A]                    evensOrOdds :: [int]
take  N []             = []                  evensOrOdds = evens
take  0 L              = []                % evensOrOdds = odds
take (N + 1) [X | Xs] = [X | take N Xs]

evens :: [int]                          evens1 :: int -> [int]
evens = [2 | evens1 4]                  evens1 N = [N | evens1 (N + 2)]

odds :: [int]                           odds1 :: int -> [int]
odds  = [1 | odds1  3]                  odds1  N = [N | odds1  (N + 2)]
```

The right version of this program is the one including the commented (`%`) program rule `evensOrOdds = odds`. We suppose that the user tries to execute the goal `take 3 evenOrOdds == L` in order to obtain in `L` the list of the first three even numbers and the list of the first three odd numbers in a non-deterministic way. The user executes it but finds that the system gives only one answer: the list of the first three even numbers {`L -> [2,4,6]`}. After running the missing answers debugging tool, a new window with the graphical representation of the *CT* pops

Fig. 2. Debugging session guided by the *top-down* strategy.

up (see **Fig. 1**). The graphical interface has been implemented in *Java* following the line of previous works [9,10,11].

At this point an incomplete function `evensOrOdds` can be found by inspecting the *CT* and noticing that the *aca* corresponding to the function call `take 3 evensOrOdds ==> [2:4:6:[]]` has only one of the two expected lists as computed answers, so that the other one is missing and then there exists a mistake in the definition of the function `evensOrOdds`. This process is easily performed in this situation because the set of nodes to be inspected is relatively small. However, inspecting bigger sets might be a time consuming and difficult task. In these cases, the debugging system provides several implemented *algorithmic debugging strategies* (see [13] for more details) to guide the diagnosis session until a buggy node is found. For example, **Fig. 2** illustrates the interactive application of the so-called *top-down* strategy to identify a buggy node.

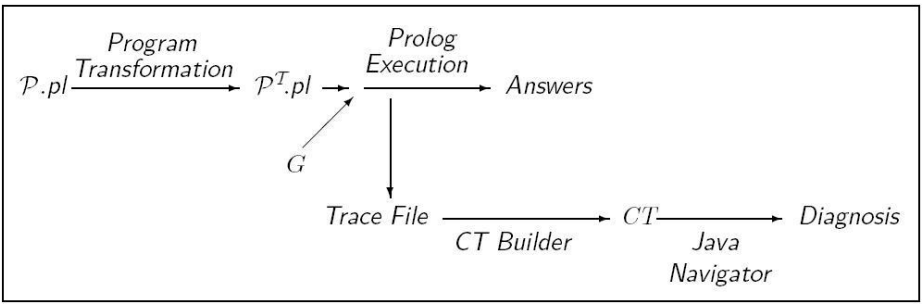

Fig. 3. The missing answers debugger implementation schema.

4 Implementation of the Declarative Debugger

In this section we give technical details of the implementation of our declarative debugger for missing answers in the $\mathcal{TOY}$ system. The current implementation of the debugger is also available at the Web page. The compiler of $\mathcal{TOY}$ is written in *Prolog* and converts a source program $\mathcal{P}.toy$ into an equivalent *Prolog* program $\mathcal{P}.pl$ which will be executed by the underlying *Prolog* system. The main novelty of our work has been the integration of the debugger as a *stand-alone module*, without modifying the rest of the system. To perform this task, we use a declarative view of the trace of the computation to generate the CT, rather than only a more classical program transformation approach implemented in previous works on wrong answers in lazy *CFLP* systems (see, [9,11]). More precisely, according to **Fig. 3**, the debugging process is performed in four steps:

(1) The intermediate file $\mathcal{P}.pl$ modifies each user defined function. This modification doesn't change the semantic of the function, but after any successful function call the trace now displays the values of the arguments and result. The execution of this modified program $\mathcal{P}^{\mathcal{T}}.pl$ will generate a text file containing the trace of the computation.

(2) The goal G is evaluated again but now w.r.t. the modified program, retrieving all the answers until the computation ends. This evaluation generates a file with the trace of the computation described before. In the next steps the debugger works with this trace file, and it is independent of how the actual trace was generated. This makes our method easy to port to other functional-logic languages like *Curry* (see [14]), because just a little modification on the way that this trace file is generated should be needed.

(3) The trace file is processed, removing unnecessary information and preparing it for the generation of the CT by the *CT Builder* module of the $\mathcal{TOY}$ system.

(4) The CT is reconstructed processing all the facts contained in the simplified trace file. This stage is performed interpreting the trace w.r.t. the declarative model of evaluation of $\mathcal{TOY}$. This stage should also be modified when implementing this method for another similar *FLP* system.

Once the CT is reconstructed, it is ready to be printed in a text file and ready to be connected to the graphical module that will show it to the user.
Using a declarative and simplified version of the trace of the computation to generate the CT has allowed us to reach our goal of developing a debugger without modifying the core modules of the $\mathcal{TOY}$ system. Also, we have been able to develop a debugger that would be relatively easy to translate to other functional and logic systems. However, the main problem with this approach is the lack of efficiency and scalability. The process of reconstructing the CT from a suitable view of the trace of the program is expensive both in terms of memory and time. In the second phase of the process described above, a trace file with a *Prolog* fact corresponding to each input and output for every function call is generated. During the evaluation of even small test programs, hundreds of function calls are performed, mainly due to *backtracking* and because of that hundreds of *Prolog* facts are generated. The real problem is that, in the fourth step, this text file is analyzed and the whole CT is stored in memory while this process occurs. This huge trace would also lead to a CT with a lot of nodes, and so that the memory space required and the time consumed to perform all the tasks involving that computation tree would be excessive for "real life" programs. Currently, it is not possible to build a small portion of the computation tree using a selected fragment of the trace because the tree is correctly built only when all the trace facts have been sequentially processed. We are working on improvements of the system trying to reduce this memory and time cost. Our main goal is to keep dinamically in memory only the portion of the computation tree displayed at each moment from the corresponding declarative view of the simplified trace.

In spite of this computational overhead, our current system can cope with CTs containing thousands of nodes, which is enough for medium size computations. Moreover, the system works reasonably well in cases where the goal's search space is relatively small, and we believe that working with such goals can be useful for detecting many programming bugs in practice. We believe that our debugger offers better facilities for CT simplification and navigation, which means a crucial advantage in CTs with a large number of nodes, where *top-down* navigation produces too many (maybe complex) questions to the user (as explained in [13]). We are improving the implementation of the system, integrating techniques for simplifying large and intricate *aca*s, asking the user for a concrete missing instance of the initial goal and starting a diagnosis session for the instantiated goal.

5 Conclusions, Related and Future Work

In this short paper we have described on-going work on the system description of a graphical environment for finding missing answers and incomplete program rules in lazy functional-logic programs. The debugger is equipped with a graphical user interface and has been implemented for the *FLP* system $\mathcal{TOY}$, although the same approaches and ideas can be used for debugging similar languages such as *Curry* or *CiaoPP* [15]. In comparison to the traditional *top-down* declarative debuggers, we give more support for avoiding the complexity of oracle questions.

In contrast to other debugging tools (as e.g. the visual debugger for *Mercury* [16]), our debugger is an *off-line* tool: the *CT* must be completely generated before it can be displayed and navigated.

As future work we plan an extension of our current debugger supporting the declarative diagnosis of both *wrong* and *missing* answers. Moreover, we plan to investigate and implement extensions of the debugger to support *constraint-based computations* in constraint functional and logic programming following the line of [8]. Hopefully, this eventually helps us to evaluate the debugger on practical applications. We also plan to implement and evaluate alternative *search strategies* for the navigation phase. Other extensions can be done by studying the integration of *assertion based methods* for declarative debugging [17].

References

1. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, USA (1983)
2. Ferrand, G., Lesaint, W., Tessier, A.: Towards declarative diagnosis of constraint programs over finite domains. ArXiv Computer Science e-prints (2003)
3. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. J. Funct. Program. **11**(6) (2001) 629–671
4. Nilsson, H., Sparud, J.: The evaluation dependence tree as a basis for lazy functional debugging. Autom. Softw. Eng. **4**(2) (1997) 121–150
5. Pope, B., Naish, L.: Practical aspects of declarative debugging in haskell 98. In: PPDP'03, ACM (2003) 230–240
6. Tessier, A., Ferrand, G.: Declarative diagnosis in the *CLP* scheme. In: Analysis and Visualization Tools for Constraint Programming. Volume 1870 of LNCS., Springer (2000) 151–174
7. López-Fraguas, F.J., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: RTA'99. Volume 1631 of LNCS., Springer (1999) 244–247
8. Caballero, R., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: Declarative diagnosis of missing answers in constraint functional-logic programming. In: FLOPS'08. Volume 4989 of LNCS., Springer (2008) 305–321
9. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: WCFLP'05, ACM (2005) 8–13
10. Caballero, R., Rodríguez-Artalejo, M.: $\mathcal{DDT}$: A declarative debugging tool for functional-logic languages. In: FLOPS. Volume 2998 of LNCS. (2004) 70–84
11. del Vado-Vírseda, R.: A logical framework for debugging in declarative constraint programming. Electr. Notes Theor. Comput. Sci. **256** (2009) 119–135
12. Naish, L., Barbour, T.: A declarative debugger for a logical-functional language. DSTO General Document **5(2)** (1995) 91–99
13. Silva, J.: A comparative study of algorithmic debugging strategies. In: LOPSTR'06. Volume 4407 of LNCS., Springer (2006) 143–159
14. Hanus, M.: *Curry*: An integrated functional logic language (version 0.8.2 of march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~curry` (2006)
15. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Abstract verification and debugging of constraint logic programs. In: IWCSCLP'02. Volume 2627 of LNCS., Springer (2002) 1–14
16. Cameron, M., de la Banda, M.G., Marriott, K., Moulder, P.: Vimer: A visual debugger for mercury. (2003) 56–66

17. Drabent, W., Nadjim-Tehrani, S., Maluszynski, J.: The use of assertions in algorithmic debugging. (1988) 573–581

Efficient and Compositional Higher-Order Streams

Gergely Patai

Budapest University of Technology and Economics, Budapest, Hungary
patai@iit.bme.hu

Abstract. Stream-based programming has been around for a long time, but it is typically restricted to static data-flow networks. By introducing first-class streams that implement the monad interface, we can describe arbitrary dynamic networks in an elegant and consistent way using only two extra primitives besides the monadic operations. This paper presents an efficient stream implementation and demonstrates the compositionality of the constructs by mapping them to functions over natural numbers.

1 Introduction

One of the major advantages offered by pure functional programming is the possibility of equational reasoning in software development. This is achieved by enforcing referential transparency, which precludes the use of observable mutable state. However, while pure functions can easily describe the transformation of one data structure into another, interactive and embedded applications have to deal with input and output living in a time domain and describe temporal as well as functional dependencies. In practice, this means that the state of the computation has to be maintained and regularly projected on the output, be it a user interface, an actuator or a consumer process.

Stream-based programming is an approach that avoids the introduction of a monolithic world state. The basic idea is that every variable in the program represents *the whole lifetime* of a time-varying value. For instance, an expression like $x + y$ might describe a combinational network that takes two input streams and outputs a stream defined by their point-wise sum. On the implementation level, x and y can be represented with mutable variables, but the stream concept allows us to compose them as pure values.

Traditional stream-based languages like Lustre [8] allow us to describe static data-flow networks and compile them into efficient loops. In many cases this is not enough. As soon as we want to describe dynamically reconfigurable systems or a collection of entities that changes over time – and such a need can come up even in a relatively simple system like a sensor network node –, we need more expressive power.

Functional reactive programming is essentially an extension of the stream-based approach that adds higher-order constructs in some form. Its first incarnation, Fran [6], introduced time-varying values as first-class entities. Probably

the most important lesson of Fran was the realisation that the start times of streams must be treated with care. We can either decide to work with global time (fix the start time of every stream to the beginning of execution) or local time (have every stream count time from its own creation). Global-time streams are naturally composable, but it is easy to create space and time leaks with them. For instance, if we are allowed to synthesise the integral of an input in the middle of the execution, we have to keep all the past input in the memory (space leak), and the freshly created integrator has to catch up immediately by summing over all that past (time leak). Local-time semantics is arguably more intuitive and does not suffer from this problem, but it can easily break referential transparency if introduced naively: the meaning of $x + y$ can change if x or y is translated in time.

This paper presents an approach that unites the advantages of the two worlds: referentially transparent ('compositional') higher-order streams without space or time leaks ('efficient'). First we will discuss the original problem in more detail (Section 2), then we will derive a minimal set of combinators to describe a wide range of dynamic networks that can be potentially realised without a major performance hit (Section 3). The power of the resulting interface is illustrated through a longer example (Section 4). Afterwards, we can move to the implementation side; we will see the difficulties of a pure approach (Section 5), and outline an impure runnable implementation of the idea, making some practical adjustments to the interface on the way (Section 6). We conclude the discussion with a short evaluation (Section 7) and comparison to related work (Section 8).

2 Problems with Higher-Order Streams

For our purposes, a stream is an object that can be observed through two functions, *head* and *tail*, which extract its first element and its immediate continuation, respectively. If the type of the elements is a, the type of the stream is referred to as *Stream* a.

Streams are applicative functors [11], i.e. we can define two constructors, *pure* and $\circledast$, to describe the stateless combination of streams. In order to construct arbitrary causal static networks, we need only one more building block: a unit delay with initialisation. We will refer to it as *cons*, since cons streams are the trivial implementation of this interface. Feedback can be directly expressed through value recursion, so we do not need a specific fixed-point operator.

The exact meaning of the constructors can be defined coinductively through their interaction with the destructors, which is shown on Figure 1; we denote a stream s as $\langle s_0\ s_1\ s_2\ s_3 \dots \rangle$. If s is of type *Stream* a, then s_i is of type a.

Things get more interesting if we want to add dynamism to the network structure. If we can define an operation to flatten higher-order streams, we can model the changing topology by a stream that delivers a potentially changing part of the network at every point of time. The flattening operation turns the higher-order stream into the actual dynamic network. Not surprisingly, in order to end up with intuitive behaviour, the combinator we are looking for should

$$\langle x\ s_0\ s_1\ s_2\ s_3 \dots \rangle \qquad \langle x\ x\ x\ x\ x \dots \rangle \qquad \langle (f_0\ x_0)\ (f_1\ x_1)\ (f_2\ x_2)\ (f_3\ x_3) \dots \rangle$$

$$cons\ x\ s \qquad pure\ x \qquad f \circledast x$$

$$\begin{array}{lll} head\ (cons\ x\ s) \equiv x & head\ (pure\ x) \equiv x & head\ (f \circledast x) \equiv (head\ f)\ (head\ x) \\ tail\ (cons\ x\ s) \equiv s & tail\ (pure\ x) \equiv pure\ x & tail\ (f \circledast x) \equiv tail\ f \circledast tail\ x \end{array}$$

Fig. 1. First-order stream constructors

obey the laws of the monadic *join* operation. The laws basically state that in case we have several layers, it does not matter which order we break them down in, and *join* commutes with point-wise function application (*fmap*, where $fmap\ f\ s \equiv pure\ f \circledast s$), so it does not matter either whether we apply a stateless transformation before or after collapsing the layers.

Finding the appropriate *join* operation is straightforward if we build on the isomorphism between the types $Stream\ a$ and $\mathbb{N} \to a$. All we need to do is adapt the monad instance of functions with a fixed input type (aka the reader monad) to streams, where *head* corresponds to applying the function to zero, while *tail* means composing it with *succ*. Just as the other constructors, *join* can be defined through its observed behaviour:

$$\begin{array}{l} head\ (join\ s) \equiv head\ (head\ s) \\ tail\ (join\ s) \equiv join\ (fmap\ tail\ (tail\ s)) \end{array}$$

In other words, $join\ s = \langle s_{00}\ s_{11}\ s_{22}\ s_{33} \dots \rangle$, the main diagonal of the stream of streams s.

This is all fine, except for the sad fact that the above definition, while technically correct, is completely impractical due to efficiency reasons: the n^{th} sample takes n^2 steps to evaluate, since every time we advance in the stream, we prepend a call to *tail* for every future sample.

The problem is that each sample can potentially depend on all earlier samples (cf. recursive functions over $\mathbb{N}$), so there are no shortcuts to take advantage of in general. The only optimisation that can help is special-casing for constant streams. However, it would only make *join* efficient in trivial cases that do not use the expressive power of the monadic interface, hence it is no real solution. Instead, we have to find a different set of constructors, which can only generate structures where the aforementioned shortcut is always possible.

3 Doing without Cons

First of all, we will assume that streams are accessed sequentially. The key to finding a leak-free constructor base is realising that we do not need to sample the past of the newly created streams. We want to ensure that whenever a new stream is synthesised, its origin is defined to be the point of creation. However, in order to maintain referential transparency, we will have to work with global time.

The basic idea is to map local-time components on the global timeline, effectively keeping track of their start times.

The monadic operations are all safe, because they are stateless, hence time invariant. The cause of our problems is *cons*, which has no slot to encode its start time, which is therefore implicitly understood to be zero. But this means that there is nothing to stop us from defining a stateful stream that starts in the past, so we will have to disallow *cons* altogether and look for a suitable alternative.

Let us first think about how starting time affects streams. For simplicity, we will step back and represent streams as functions of time, i.e. $\mathbb{N} \to a$, and try to derive a sensible interface using *denotational design* [4].

By choosing a representation we also inherit its class instances, so the monadic operations are readily available. Let us introduce a local-time memory element called *delay*. In order to express its dependence on the start time, we can simply pass that time as an extra argument:

$$
\begin{aligned}
&\mathit{delay}\ x\ s\ t_{start}\ t_{sample} \\
&\quad \mid t_{start} \equiv t_{sample} = x \\
&\quad \mid t_{start} < t_{sample} = s\ (t_{sample} - 1) \\
&\quad \mid \mathit{otherwise} \qquad = \mathit{error}\ \texttt{"Premature sample!"}
\end{aligned}
$$

There is a fundamental change here: *delay x s* – unlike *cons x s* – is not a stream, but a *stream generator*. In this model, stream generators are of type $\mathbb{N} \to a$, functions that take a starting time and return a data structure that might hold freshly created streams as well as old ones. Why not limit them to $\mathbb{N} \to \mathit{Stream}\ a$? Because that would limit us to generate one stream at each step. A stream that carries a list of values is not equivalent to a list that contains independent streams, since the former solution does not allow us to manage the lifetimes of streams separately. There is no way to tell if no-one ever wants to read the first value of the list, therefore we would not be able to get rid of it, while independent streams could be garbage collected as soon as all references to them are lost.

Since the types of streams and stream generators coincide while the arguments of the functions have different meanings (sampling time and starting time, respectively), let us introduce type synonyms to prevent confusion:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Stream}\ a = \mathbb{N} \to a \\
&\textbf{type}\ \mathit{StreamGen}\ a = \mathbb{N} \to a
\end{aligned}
$$

Using the new names, the type of *delay* can be specified the following way:

$$\mathit{delay} :: a \to \mathit{Stream}\ a \to \mathit{StreamGen}\ (\mathit{Stream}\ a)$$

It is clear from the above definition that a *delay* is not safe to use if its start time is in the future, therefore we should not allow generators to be used in arbitrary ways. The only safe starting time is the smallest possible value, which will lead us to the sole way of directly executing a stream generator: passing it zero as the start time.

$start :: StreamGen\ (Stream\ a) \to Stream\ a$
$start\ g = g\ 0$

While there is no immediate need to restrict its output type, we basically want a function that is used only once at the beginning to extract the top-level stream. This makes our interface as tight as possible.

Of course, *start* is not sufficient by itself, since it does not give us any means to create streams at any point later than the very first sampling point. We need another combinator that can extract generators in a disciplined way, ensuring that no unsafe starting times can be specified. The most basic choice is simply extracting a stream of generators by passing each the current sampling time:

$generator :: Stream\ (StreamGen\ a) \to Stream\ a$
$generator\ g\ t_{sample} = g\ t_{sample}\ t_{sample}$

As it turns out, in our model *generator* happens to coincide with *join* for functions. However, this does not hold if *Stream* and *StreamGen* are different structures, so *generator* remains an essential combinator.

Of course, if we look at it from a different direction, we can say that stream generators can also be considered streams. In this case, *start* is simply equivalent to *head* (which also hints at the fact that *start* is not a constructor), *generator* is the same as *join*, and *delay* constructs a stream of streams. We also inherit the fixed-point combinator *mfix* [7] from the reader monad, which will be necessary to define feedback loops. In the end, we can reduce the combinator base required to work with higher-order dataflow networks to the one on Figure 2; we use *return* and $\gg\!=$ in the minimal base, since they can express the applicative combinators as well as *join*.

Even though the four-combinator base is quite elegant, the distinction between streams and stream generators is still useful. The main difference between *join* (on streams) and *generator* is that the latter is used to create new streams, while the former can only sample existing streams. Also, we will see that *mfix* for stream generators allows us to define streams in terms of each other (e.g. express the circular dependency between position, velocity and acceleration in the context of spring motion), while *mfix* for streams has no obvious practical application.

4 A Motivating Example

Let us now forget about the representations we chose and keep only the interfaces. These are quite small: both streams and stream generators are *MonadFix* instances, and we have *delay*, *generator* and *start* to work with.

We saw that *delay* is the only operation that lives in the *StreamGen* monad, and it allows us to create a new stateful stream every time the monad is extracted. As a simple example, we can start by defining a generic stateful stream (Ⓢ stands for infix *fmap*).

$stateful :: a \to (a \to a) \to StreamGen\ (Stream\ a)$
$stateful\ x_0\ f = mfix\ \$\ \lambda str \to delay\ x_0\ (f\ Ⓢ\ str)$

$$\begin{array}{l}\langle\langle\ x\ s_0\ s_1\ s_2\ s_3\ldots\rangle \\ \langle\ \bot\ x\ s_1\ s_2\ s_3\ldots\rangle \\ \langle\ \bot\ \bot\ x\ s_2\ s_3\ldots\rangle \\ \langle\ \bot\ \bot\ \bot\ x\ s_3\ldots\rangle \\ \ldots\qquad\qquad\rangle\end{array} \qquad \langle x\ x\ x\ x\ldots\rangle \qquad \begin{array}{l}\langle y_{00}\ y_{11}\ y_{22}\ y_{33}\ y_{44}\ldots\rangle \\ \textbf{where}\ y_i = f\ s_i\end{array}$$

$$\mathit{delay}\ x\ s \qquad \mathit{return}\ x \qquad s \gg\!= f$$

$$\begin{array}{l}\langle(\mathit{fix}\ (\mathit{nth}\ 0 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 1 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 2 \circ f))\ (\mathit{fix}\ (\mathit{nth}\ 3 \circ f))\ldots\rangle \\ \quad \textbf{where}\ \mathit{nth}\ 0\ s = \mathit{head}\ s \\ \qquad\qquad \mathit{nth}\ n\ s = \mathit{nth}\ (n-1)\ (\mathit{tail}\ s)\end{array}$$

$$\mathit{mfix}\ f$$

$$\begin{array}{ll}\mathit{head}\ (\mathit{delay}\ x\ s) \equiv \mathit{cons}\ x\ s & \mathit{head}\ (\mathit{return}\ x) \equiv x \\ \mathit{tail}\ (\mathit{delay}\ x\ s) \equiv \mathit{fmap}\ (\mathit{cons}\ \bot)\ (\mathit{delay}\ x\ (\mathit{tail}\ s)) & \mathit{tail}\ (\mathit{return}\ x) \equiv \mathit{return}\ x\end{array}$$

$$\begin{array}{ll}\mathit{head}\ (s \gg\!= f) \equiv \mathit{head}\ (f\ (\mathit{head}\ s)) & \mathit{head}\ (\mathit{mfix}\ f) \equiv \mathit{fix}\ (\mathit{head} \circ f) \\ \mathit{tail}\ (s \gg\!= f) \equiv \mathit{tail}\ s \gg\!= (\mathit{tail} \circ f) & \mathit{tail}\ (\mathit{mfix}\ f) \equiv \mathit{mfix}\ (\mathit{tail} \circ f)\end{array}$$

Fig. 2. Higher-order stream constructors

Simply put, a stream generated by *stateful* $x_0\ f$ starts out as x_0, and each of its subsequent outputs equals f applied to the previous one. Note that we could not define *stateful* using direct recursion, since *delay* adds an extra monadic layer, so we had to rely on *StreamGen* being *MonadFix*.

Let us run a little test. This is the only place where we rely on functions being our representation, since we pass the generated stream to *map*.

```
strtest :: StreamGen (Stream a) → [a]
strtest g = map (start g) [0..15]
> strtest $ stateful 2 (+3)
[2,5,8,11,14,17,20,23,26,29,32,35,38,41,44,47]
```

This looks promising, but a more complex example involving higher-order streams would be more motivating. Let us create a dynamic collection of countdown timers, where each expired timer is removed from the collection. First we will define named timers:

```
countdown :: String → Int → StreamGen (Stream (String, Maybe Int))
countdown name t = do
  let tick prev = do {t ← prev; guard (t > 0); return (t − 1)}
  timer ← stateful (Just t) tick
  return ((,) name <$> timer)
> strtest $ countdown "foo" 4
[("foo", Just 4), ("foo", Just 3), ("foo", Just 2), ("foo", Just 1),
 ("foo", Just 0), ("foo", Nothing), ("foo", Nothing), ...]
```

Next, we will define a timer source that takes a list of timer names, starting values and start times and creates a stream that delivers the list of new timers at every point. Naturally, it would be more efficient to consume the original list as we advance in the stream, but mapping over a simple counter will do for now.

```
timerSource :: [(String, Int, Int)] →
                  StreamGen (Stream [Stream (String, Maybe Int)])
timerSource ts = do
  let gen t = mapM (uncurry countdown) newTimers
        where newTimers = [(n, v) | (n, v, st) ← ts, st ≡ t]
  cnt ← stateful 0 (+1)
  return $ generator (gen ⓢ cnt)
```

Now we need to encapsulate the timer source stream in another stream expression that takes care of maintaining the list of live timers. Since working with dynamic collections is a recurring task, let us define a generic combinator that maintains a dynamic list of streams given a source and a test that tells from the output of each stream whether it should be kept. We can use μ**do** expressions (a variant of **do** expressions allowing forward references) as syntactic sugar for *mfix* to make life easier.

```
collection :: Stream [Stream a] → (a → Bool) → StreamGen (Stream [a])
collection source isAlive = μdo
  coll ← liftA2 (++) source ⓢ delay [] coll'
  let collWithVals = zip ⓢ (sequence =≪ coll) ⊛ coll
      collWithVals' = filter (isAlive ∘ fst) ⓢ collWithVals
      coll' = map snd ⓢ collWithVals'
  return $ map fst ⓢ collWithVals'
```

We need recursion to define the *coll* stream as a delayed version of *coll′*, which represents its continuation. At every point of time its output is concatenated with that of the source (we need to lift the (++) operator twice in order to get behind both the *StreamGen* and the *Stream* abstraction). Then we define *collWithVals*, which simply pairs up every stream with its current output. The output is obtained by extracting the current value of the stream container and sampling each element with *sequence*. We can then derive *collWithVals′*, which contains only the streams that must be kept for the next round along with their output. By throwing out the respective parts, we can get both the final output and the collection for the next step (*coll′*).

Now we can easily finish our task:

```
timers :: [(String, Int, Int)] → StreamGen (Stream [(String, Int)])
timers timerData = do
  src ← timerSource timerData
  getOutput ⓢ collection src (isJust ∘ snd)
    where getOutput = fmap (map (λ(name, Just val) → (name, val)))
```

Let us start four timers as a test: 'a' at $t = 0$ with value 3, 'b' and 'c' at $t = 1$ with values 5 and 3, and 'd' at $t = 3$ with value 4:

$$
\begin{array}{l}
> \mathit{strtest}\ \$\ \mathit{timers}\ [(\texttt{"a"}, 3, 0), (\texttt{"b"}, 5, 1), (\texttt{"c"}, 3, 1), (\texttt{"d"}, 4, 3)] \\
[[(\texttt{"a"}, 3)], [(\texttt{"b"}, 5), (\texttt{"c"}, 3), (\texttt{"a"}, 2)], [(\texttt{"b"}, 4), (\texttt{"c"}, 2), (\texttt{"a"}, 1)], \\
\ [(\texttt{"d"}, 4), (\texttt{"b"}, 3), (\texttt{"c"}, 1), (\texttt{"a"}, 0)], [(\texttt{"d"}, 3), (\texttt{"b"}, 2), (\texttt{"c"}, 0)], \\
\ [(\texttt{"d"}, 2), (\texttt{"b"}, 1)], [(\texttt{"d"}, 1), (\texttt{"b"}, 0)], [(\texttt{"d"}, 0)], [], [], [], [], [], [], [], []]
\end{array}
$$

While it might look like a lot of trouble to create such a simple stream, it is worth noting that *timers* is defined in a modular way: the countdown mechanism that drives each timer is completely separated, and we do not even need to worry about explicitly updating the state of the timers. Similarly, the timer source is also independent of the dynamic list, and we even defined a generic combinator that turns a source of streams into a dynamic collection of streams.

A valid concern about the interface could be the ubiquitous use of lifting. There are basically two ways to deal with this problem. First of all, since streams are monads, we could simply use **do** notation to extract the current samples of streams we need and refer to those samples directly afterwards. However, this would no doubt make the code less succinct. The other option is using syntactic support for applicative functors, e.g. *idiom brackets* [11]. For instance, the Strathclyde Haskell Enhancement [10] allows us to rewrite *collection* with a somewhat lighter applicative notation:

$$
\begin{array}{l}
\mathit{collection} :: \mathit{Stream}\ [\mathit{Stream}\ a] \to (a \to \mathit{Bool}) \to \mathit{StreamGen}\ (\mathit{Stream}\ [a]) \\
\mathit{collection}\ \mathit{source}\ \mathit{isAlive} = \mu\mathbf{do} \\
\quad \mathit{dcoll}' \leftarrow \mathit{delay}\ [\,]\ \mathit{coll}' \\
\quad \mathbf{let}\ \mathit{coll} = (\!|\ \mathit{source} +\!\!+ \mathit{dcoll}'\ |\!) \\
\qquad \mathit{collWithVals} = (\!|\ \mathit{zip}\ (\mathit{sequence} =\!\!\ll \mathit{coll})\ \mathit{coll}\ |\!) \\
\qquad \mathit{collWithVals}' = (\!|\ \mathit{filter}\ \tilde{}(\mathit{isAlive} \circ \mathit{fst})\ \mathit{collWithVals}\ |\!) \\
\qquad \mathit{coll}' = (\!|\ \mathit{map}\ \tilde{}\mathit{snd}\ \mathit{collWithVals}'\ |\!) \\
\quad \mathit{return}\ (\!|\ \mathit{map}\ \tilde{}\mathit{fst}\ \mathit{collWithVals}'\ |\!)
\end{array}
$$

5 Problems with Purity

Our goal is to come up with a way to execute any higher-order stream representable by our constructors in a way that the computational effort required for a sample is at worst proportional to the size of the network. In terms of equations, this means that applying *tail* to a stream several times results in an expression that is roughly the same size as the original. This clearly does not hold according to the rules given on Figure 2, since most streams keep growing without bounds.

Thanks to the way the *delay* combinator is designed, all higher-order streams have a main diagonal that does not depend on any element on its left – that is what makes the shortcut possible in theory. This observation points into the direction of a solution where we skew every higher-order stream in a way that its main diagonal becomes its first column, as illustrated on Figure 3. However, if we want to implement this idea directly, we will quickly run into roadblocks.

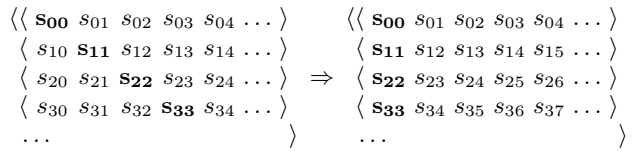

Fig. 3. Skewing higher-order streams to get an efficient *join*

The new definition of *delay* is straightforward: we simply do not add the triangular $\perp$ padding, which can be achieved by rewriting the second rule in its definition to $tail\ (delay\ x\ s) \equiv delay\ x\ (tail\ s)$. If we use *fmap*, *return* and *join* for our monadic combinator base, we will find that *join* is also easy to adapt to the skewed version: $tail\ (join\ s) \equiv join\ (tail\ s)$, since *join* now extracts the first column, shown in bold on Figure 3. This definition of *join* obeys the law of associativity, which is promising.

Somewhat surprisingly, the first bump comes when we want to describe the behaviour of *return*, which was trivial in the original model. In order to respect the monad laws, we have to ensure that if *return* is applied to a stream, then the resulting higher-order stream must contain the aged versions of the original, so the first column (which could be extracted by a subsequent *join*) is the same as the stream passed to *return*. In fact, the situation is even more complicated: the argument of *return* can be an arbitrary structure, and we have to make sure that all streams referenced in this structure are properly aged. Given a sufficiently flexible type system, the problem of *return* can be solved at a price of some extra administrative burden on the programmer. For instance, type families in Haskell can be used to define *head* and *tail* operations that traverse any structure and project every stream in the structure to its head or tail.

Unfortunately, we have two combinators, *fmap* and *mfix*, which require us to skew a higher-order stream that is produced by a function. This appears in the original rules as a composition with *tail*. The net effect of the composition is that every stream referenced in the closure is aged in each step, so in order to have a pure implementation, we would have to open up the closure, store all the streams in question alongside the function, age all these streams by applying *tail* to them in each step, and reconstruct the closure by applying the function to the aged streams.

While it is possible to go down the road outlined above, it would essentially amount to adding a layer of interpretation. However, what we are trying to do here is basically emulating mutable references, so we could as well pick the straightforward solution and use mutable variables to represent streams. The following section presents a solution in non-portable (GHC specific) Haskell.

6 Making It Run

We used the type $\mathbb{N} \to a$ to model streams. Since in the case of sequential sampling the index is implicit, we can drop the argument of the function and substitute it with side effects. This will give us the actual type:

$$\textbf{newtype}\ \mathit{Stream}\ a = S\ (\mathit{IO}\ a)\ \textbf{deriving}\ (\mathit{Functor}, \mathit{Applicative}, \mathit{Monad})$$

Every stream is represented by an action that returns its current sample. Just as it was the case in our model above, this operation has to be idempotent within each sample, and the *Stream* monad has to be commutative, i.e. it should not matter which order we sample different streams in. Both criteria are trivially fulfilled if the action has no other side effects than caching the sample after the first reading.

To be able to fill this action with meaning, we have to think about building and executing the stream network. Since *delay* is a stateful combinator, it has to be associated with a mutable variable. Delay elements are created in the *StreamGen* monad, which is therefore a natural source for these variables. But what should the internal structure of *StreamGen* look like?

When we execute the network, we have to update it in each sampling step in a way that preserves consistency. In effect, this means that no sampling action can change its output until the whole network is stepped, since streams can depend on each other in arbitrary ways. The solution is a two-phase superstep: first we go through all the variables and update them for the next round while preserving their current sample, and we discard these samples in a second sweep. We should sample the output of the whole network before the two sweeps. This means that each delay element requires the following components:

1. a mutable variable
2. a sampling action that produces its current value
3. an updating action that does not change the output of the sampling action, but creates all the data needed for the next step
4. a finalising action that concludes the update and advances the stream for the next superstep, discarding the old sample on the way

As we have seen, the sampling action is stored in the stream structure, and it obviously has to contain a reference to the variable, so the first two items are catered for. The updating and finalising actions have to be stored separately, because they have to be executed regardless of whether the stream was sampled in the current superstep or not – we certainly do not want the behaviour of any stream to be affected by how we observe it.

In short, we need to maintain a pool for these update actions. We have to make sure that whenever all references to a stream are lost, its update actions are also thrown away. This is easy to achieve using weak references to the actions, where the key is the corresponding stream. It should be noted that weak references constitute the non-portable bit, and they also force us into the IO monad even if our stream network has no external input. Summing up, our *update pool* is a list

of weak references that point us to the updating and finalising actions required for the superstep.

```
type UpdatePool = [Weak (IO (), IO ())]
```

The next step is to integrate the update pool with the sampling actions. One might first think that *StreamGen* should be a state monad stacked on top of IO, with the pool stored in the state. However, after creating the executable stream structure, it does not live in the *StreamGen* monad any more, therefore there is no way to thread this state through the monads extracted by *generator*. We have no other choice but to provide a mutable reference to the pool associated with the network the generator lives in.

```
newtype StreamGen a = SG{unSG :: IORef UpdatePool → IO a}
```

This is in fact a reader monad stacked on top of IO, but we will just use a 'raw' function for simplicity; we can easily define the equivalent *Monad* and *MonadFix* instances by emulating the behaviour of the reader monad transformer.

In order to be able to embed the stream framework in our applications, we need a function to turn a *StreamGen*-supplied stream into an IO computation that returns the next sample upon each invocation. All we need to do is create a variable for the update pool (initialised with an empty list), extract the top-level signal from its generator and assemble an IO action that performs the superstep as described above. The type of *start* is changed to reflect its new meaning.

```
start :: StreamGen (Stream a) → IO (IO a)
start (SG gen) = do
  pool ← newIORef []
  S sample ← gen pool
  return $ do
    let deref ptr = (fmap ∘ fmap) ((,) ptr) (deRefWeak ptr)
    ‡ Extracting the top-level output
    res ← sample
    ‡ Extracting the live references and throwing out the dead ones
    (ptrs, acts) ← unzip ∘ catMaybes $ (mapM deref =≪ readIORef pool)
    writeIORef pool ptrs
    ‡ Updating variables
    mapM_ fst acts
    ‡ Finalising variables
    mapM_ snd acts
    return res
```

The *deRefWeak* function returns *Nothing* when its key is unreachable, so we have to pair up every reference with the data it points to in order to be able to tell which ones we need to keep. The rest is straightforward: we extract the current output then perform the two sweeps. But how do the updating actions look like inside?

First of all, the mutable variable has to hold a data structure capable of encoding the phases. Since simple delays are the only stateful entities we have to worry about, the structure is straightforward.

$$\textbf{data}\ \mathit{Phase}\ a = \mathit{Ready}\{\,\mathit{state} :: a\,\} \mid \mathit{Updated}\{\,\mathit{state} :: a, \mathit{cache} :: a\,\}$$

A stream is either *Ready*, waiting to be sampled, or *Updated*, holding its next state and remembering its current sample. It is definitely *Updated* after executing the first sweep action, and definitely *Ready* after finalisation. In the case of *delay* we also have to make sure that the delayed signal is not sampled before the update phase, otherwise we would introduce an unnecessary dependency into the network that would cause even well-formed loops to be impossible to evaluate.

While the full implementation cannot fit in the paper, it is worth looking at the internals of at least one of the primitives. The simpler one is *delay*, where sampling is trivial and updating triggers a sampling on the delayed stream. It is advisable to force the evaluation of this sample, otherwise huge thunks might build up if a stateful stream is not read by anyone for a long time.

```
delay :: a → Stream a → StreamGen (Stream a)
delay x0 (S s) = SG $ λpool → do
  ref ← newIORef (Ready x0)
  let upd = readIORef ref >>= λv → case v of
        Ready x → s >>= λx' → x' `seq` writeIORef ref (Updated x' x)
        _       → return ()
      fin = readIORef ref >>= λ(Updated x _) → writeIORef ref $! Ready x
      str = S $ readIORef ref >>= λv → case v of
        Ready x      → return x
        Updated _ x → return x
  updateActions ← mkWeak str (upd, fin) Nothing
  modifyIORef pool (updateActions:)
  return str
```

As for *generator*, even though it is stateless, it has to know about the update pool, since its job is to populate it from time to time. Consequently, it has to live in *StreamGen*, just like *delay*. Also, it is a good idea to cache the result of its first sampling, otherwise it would create the same streams over and over, as many times as it is referenced. While this would not affect the output, it would certainly harm performance. In terms of the implementation this means that *generator* requires a variable just like *delay*. Its final type is the following:

```
generator :: Stream (StreamGen a) → StreamGen (Stream a)
```

Upon the first sampling it executes the current snapshot of the monad in its associated stream by passing it the reference to the update pool, and stores the result by flipping the state to *Updated*, keeping the state $\bot$ all the time.

There are two problems left. First of all, since we have not considered efficiency issues during the design phase, we left a source of redundant computations in the

system. If the result of an applicative operation is used in more than one place, it will be recalculated each time it is requested (e.g. the *collWithVals′* stream in the *collection* function, Section 4), because functions are not memoised by default. We can overcome this problem by introducing a third primitive called *memo*, which is observationally equivalent to *return* within *StreamGen*, but it adds a cache to any stream we pass to it.

$$memo :: Stream\ a \to StreamGen\ (Stream\ a)$$

The last issue is embedding streams in the real world. While we already have a way to read the output of a data-flow network, we cannot feed values into it. Fortunately, there is a simple standard solution in reactive programming frameworks: we can define a stream whose current reading can be explicitly set from outside using an associated sink action:

```
external :: a → IO (Stream a, a → IO ())
external x = do
  ref ← newIORef x
  return (S (readIORef ref), writeIORef ref)
```

An *external* stream is basically equivalent to a constant if we look at it from inside the network, therefore it does not have to be connected to the update mechanism in any way.

7 Closing Thoughts

A major advantage of using self-contained actions for sampling and updating the individual network nodes is that part of the evaluator is trivial to parallelise. We could introduce an update pool for each core, and execute actions in parallel, only synchronising at the end of each update sweep. The only change needed to make the code thread safe is to introduce an MVar (a variable protected with a mutex) to store actions generated during the update, and distribute these actions among the pools. As for the sampling phase, we would need to track the dependencies of the streams and possibly work in a bottom-up fashion as much as possible.

The biggest problem with the system is that it keeps updating streams even if they are not referenced any more until the garbage collector physically gets rid of them. The solution is not clear yet, and it might require more direct runtime support than general weak references.

The code is available in executable form in the experimental branch of the Elerea library[1] [12], which is available through *cabal-install*. For historical reasons, the library uses the name *Signal* instead of *Stream*, but the code is otherwise identical.

[1] see the `FRP.Elerea.Experimental.Simple` module

8 Related Work

While first-order stream-based languages are well understood, there has been comparatively little effort going into introducing first-class streams. Functional reactive programming, pioneered by Fran [6], brought a change in this regard, but the question of start times was ignored first, leading to great confusion about the interface. Also, FRP systems never had a focus on providing general higher-order constructs, and efficiency issues with Fran-like systems (besides the above mentioned confusion) led to Yampa [3], which is essentially a first-order stream library amended with switching combinators to describe dynamic networks. However, fitting the switching combinators into the arrow framework used by Yampa is not without problems, e.g. they have no place in the *causal commutative normal form* [9] of a signal function. Reactive [5] is a recent reformulation of Fran that fixes the starting time of all varying quantities to zero, therefore its general higher-order capabilities (i.e. monad instances of behaviours) are of little use in practice. FrTime [2] relies on a specialised evaluator to be able to use signals inside expressions, and it uses lexical scope to determine starting time. Unfortunately, this approach cannot be used in the presence of optimisations like let-floating. The previous version of the Elerea library supports higher-order streams by maintaining the original call graph of the program and adding a layer of interpretation. It differs from the system discussed in this paper in only updating streams that are sampled during a superstep, therefore its flattening combinator (*sampler*) breaks compositionality and does not obey the laws of *join*. In short, the present work inherited the concept of *stream generator* from the old Elerea, but the evaluation mechanism is completely reworked.

Looking at the well-known stream-based languages, Lucid Synchrone has recently been extended with some higher-order capabilities [1]. Its *every* combinator is very similar to *generator*, but it is guarded by a boolean stream that dictates when to instantiate the stateful stream function, and stores it in a memory element. In essence, it creates a piecewise function. The question of free variables in stateful definitions is solved by simply disallowing them due to the lack of clear semantics according to the authors. This restriction results in more tractable time and space behaviour, and it can be implemented without a global update pool. In particular, it does not suffer from the problem of updating lost streams, because references to generated stream functions cannot be passed around freely, so we can maintain an explicit update tree that tells us precisely which streams are still alive.

An alternative way to look at streams is modelling them as comonads [13]. However, comonads need extra machinery to be able to express applicative combinators, which come for free given the monadic interface. Also, the authors say nothing about higher-order streams, and they provide no guidance to arrive at an efficient implementation.

Finally, it is worth noting that there are several other Haskell libraries that focus on streams, but none of them support higher-order streams to the degree presented in this paper. The Stream library contains the 'leaky' *Monad* instance for cons streams that corresponds to the discussion in Section 2.

9 Future Work

Designing and structuring systems with higher-order streams is mostly uncharted land with vast areas to explore. The next step is to create a non-trivial interactive application using the library, which will give us a clearer picture of the strengths and weaknesses of this approach. Also, equational reasoning might be used to optimise stream networks at compile time, which is another interesting line of research to pursue.

References

1. Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-Order Synchronous Data-Flow Language. In *Proceedings of the 4th ACM International Conference on Embedded Software*, Pisa, Italy, 2004, pages 230–239, ACM, New York, NY, USA.
2. Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006, pages 294–308.
3. Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18. ACM Press, 2003, Uppsala, Sweden.
4. Conal Elliott. Denotational design with type class morphisms (extended version) 2009. `http://conal.net/papers/type-class-morphisms/`
5. Conal Elliott. Push-pull functional reactive programming In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Edinburgh, Scotland, 2009, pages 25–36, ACM, New York, NY, USA, 2009.
6. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, June 1997, pages 263–273.
7. Levent Erkök. Value Recursion in Monadic Computations. PhD Dissertation, Oregon Graduate Institute School of Science Engineering, OHSU, October 2002.
8. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud. The Synchronous Data-Flow Programming Language LUSTRE. In *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991, pages 1305–1320.
9. Hai Liu, Eric Cheng, and Paul Hudak. Causal Commutative Arrows and Their Optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009, Edinburgh, Scotland, pages 35–46, ACM, New York, NY, USA, 2009.
10. Conor Mcbride. The Strathclyde Haskell Enhancement. 2009. `http://personal.cis.strath.ac.uk/~conor/pub/she/`
11. Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), pages 1–13, 2008.
12. Gergely Patai. Eventless Reactivity from Scratch. In *Draft Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages*, 2009, South Orange, NJ, USA, pages 126–140, Marco T. Morazán (ed), Seton Hall University, 2009.
13. Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In *Central European Functional Programming School*, 2006, pages 135–167, Lecture Notes in Computer Science, Springer, Berlin.

Bridging the Gap between Two Concurrent Constraint Languages *

Alexei Lescaylle, Alicia Villanueva

DSIC, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia (Spain),
alescaylle@dsic.upv.es, villanue@dsic.upv.es

Abstract. The Concurrent Constraint Paradigm (cc in short) is a simple and highly expressive formalism for modeling concurrent systems where agents execute asynchronously, interacting among them by *adding* and *consulting* constraints in a global *store*. The cc model replaces the notion of store-as-valuation with the notion of store-as-constraint. There exist several programming languages that extend the cc model by introducing a notion of time. The notion of time allows us to represent concurrent and reactive systems. The different definitions for time make each language better suited for modeling a specific kind of application (deterministic embedded systems, non-deterministic reactive systems, etc.). This paper studies the relation between the *universal timed concurrent constraint* language (utcc in short) and the *timed concurrent constraint* language (tccp). We show how utcc can be mapped into tccp by means of a transformation that preserves the original behavior. We also prove the correctness of the transformation.

1 Introduction

The *Concurrent Constraint* paradigm (cc in short) [17,18] is a simple but powerful model able to represent concurrent systems. The model is based on a partial representation of information. Differently from the classical approach where a value is assigned to each system variable (store-as-valuation), in cc a store (i.e., a conjunction of constraints) defines the (partial) information regarding the values of the system variables. The store is also the way in which agents representing the behavior of the system interact. In brief, the cc model is based on agents that update and consult some information to the store by using the *tell* and *ask* operations. The *tell* operation adds a given constraint to the store, whereas *ask* checks whether or not a given constraint is entailed by the store.

The original cc model has been extended with a notion of time in several ways in order to be able to specify systems where it is necessary to model the *time*, such as reactive systems [7] and, in general, applications with timing constraints.

* This work has been supported by the Spanish MEC/MICINN under grant TIN2007-68093-C02-02, by the Generalitat Valenciana under grant Emergentes GV/2009/024, and by the Universidad Politécnica de Valencia, program PAID-06-07 (TACPAS).

In the literature there exist several programming languages resulting from the temporal extension of cc. In this work we deal with two of them: the *Timed Concurrent Constraint* (tccp) of F. de Boer *et al.* [3], and the more recent *Universal timed concurrent constraint* (utcc) of C. Olarte and F. Valencia [16]. Although both languages are defined under the same cc paradigm, they differ both in the operators they consider and in the defined notion of time.

There are many differences between the two considered languages. We remark the most important ones which are, first, that utcc is a deterministic language whereas tccp preserves the non-determinism of cc. Second, the concurrent model used in tccp is maximal paralellism whereas utcc preserves the interleaving model for concurrency of cc. Finally, in utcc a new temporal operator is defined in order to make time pass, whereas in tccp the notion of time is implicit. This means that there is no operator for controlling when a time instant passes, but each time a consult or update of the store is executed, time passes.

The expressive power of temporal concurrent constraint languages has been previously studied [3,13,14,15,19]. In [19] it is proven that the temporal language defined in [6] embeds the synchronous data flow language Lustre [4] (restricted to finite value types) and also the asynchronous state oriented language Argos [12]. Moreover, the strong abortion mechanism of Esterel can also be encoded in this language. In [3] it is shown that the notion of maximal parallelism of tccp is more expressive than the notion of interleaving parallelism of other concurrent constraint languages. Regarding the utcc language, it is shown Turing complete in [15].

In this paper, we define a transformation from utcc programs into tccp programs that preserves the semantics of the original program. The two main challenges we had to overcome were the definition in tccp of the abstraction operator of utcc, and to mimic the notion of time of utcc in the implicit time of tccp. We are interested in comparing these two languages since both, tccp in [8,9] and utcc in [16,15], have been used to model security protocols. Since they have very different characteristics, the approaches for modeling protocols are also quite different. By transforming utcc programs into tccp programs, we are able to apply over the resulting tccp program other techniques such as the abstraction method of [2] or the interpreter presented in [11].

The paper is structured as follows. Section 2 presents the utcc and tccp languages. The intuition of the transformation from utcc into tccp is given in Section 3. Then, Section 4 describes the transformation process, and the correctness of our proposal. Finally, Section 5 concludes and gives some directions for future work.

2 The utcc and tccp languages

All the languages from the cc paradigm are defined over an underlying *constraint system*. In this section, we introduce the notion of constraint system. Thereafter, we present the utcc and tccp languages.

2.1 The Constraint System

The cc framework is parametric w.r.t. a constraint system which states the constraints that can be used and the entailment relation among them. In this paper we assume that the two languages share the same constraint system.

Let us recall the definition of constraint system of [16]. A constraint system $\mathcal{C}$ can be represented as the pair (Σ, Δ) where Σ is a signature of function and predicate symbols, and Δ is a first-order theory over Σ. Let $\mathcal{L}$ be the first-order language underlying $\mathcal{C}$ with a denumerable set of variables $Var = \{x, y, \ldots\}$, and logic symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$, *true* and *false*. The set of constraints $\mathcal{C} = \{c, d, \ldots\}$ are terms over $\mathcal{L}$. We say that c entails d in Δ, written $c \vdash_\Delta d$, iff $c \Rightarrow d \in \Delta$, in other words, iff $c \Rightarrow d$ is true in all models of Δ. In the following we use $\vdash$ instead of $\vdash_\Delta$ when confusion is not possible. For operational reasons, $\vdash$ is often required to be decidable.

We use $\vec{t}$ for a sequence of terms $t_1, \ldots, t_n$ with length $|\vec{t}| = n$. If $|\vec{t}| = 0$ then $\vec{t}$ is written as ϵ. We use $c[\vec{t}\backslash\vec{x}]$, where $|\vec{t}| = |\vec{x}|$ and x_i's are pairwise distinct, to denote c with the free occurrences of x_i replaced with t_i. The substitution $[\vec{t}\backslash\vec{x}]$ is similarly applied to other syntactic entities.

2.2 Universal Timed Concurrent Constraint Language

The Universal Timed Concurrent Constraint Language (utcc in short) [16] extends the deterministic timed language of [6] (called tcc) to model mobile behavior. It introduces the parametric ask constructor $(\mathsf{abs}\,\vec{x}; c)\,A$, replacing the native ask operation of tcc. The novelty is that the parametric ask not only consults whether or not a condition holds, but also binds the variables $\vec{x}$ in A to the terms that make c true. As in the tcc language, the notion of time is modeled as a sequence of *time intervals*. During each time interval, some *internal steps* are executed until the process reaches a resting point. Let us show the syntax of the language.

Definition 1 (utcc syntax [16]). *Processes $A, B, \ldots$ in utcc are built from constraints in the underlying constraint system $\mathcal{C}$ as follows:*

$$A, B ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid A \parallel B \mid (\mathsf{local}\,\vec{x}; c)\,A \mid \mathsf{next}\,A \mid \mathsf{unless}\,c\,\mathsf{next}\,A \mid !A \mid \\ \mid (\mathsf{abs}\,\vec{x}; c)\,A$$

with the variables in $\vec{x}$ being pairwise distinct.

Intuitively, skip does nothing; $\mathsf{tell}(c)$ adds c to the shared store. $A \parallel B$ denotes A and B running in parallel (interleaving) during the current time interval whereas $(\mathsf{local}\ \vec{x}; c)\ A$ *binds* the set of fresh variables $\vec{x}$ in A. Thus, $\vec{x}$ are local to A under a constraint c. The *unit-delay* $\mathsf{next}\,A$ executes A in the next time interval. The *time-out* $\mathsf{unless}\,c\,\mathsf{next}\,A$ executes A in the next time interval iff c is not entailed at the resting point of the current time interval. These two processes explicitly control the time passing. The *replication* operator $!A$ is equivalent to execute $A \parallel \mathsf{next}\,A \parallel \mathsf{next}^2\,A \parallel \ldots$, i.e., unboundly many copies of A, one at a time. Finally, $(\mathsf{abs}\,\vec{x}; c)\,A$ executes $A[\vec{t}\backslash\vec{x}]$ in the current time interval for *each*

term $\vec{t}$ such that the condition $c[\vec{t}\backslash\vec{x}]$ holds in the store. $\mathsf{when}\, c\, \mathsf{do}\, A$ is used for the empty abstraction $(\mathsf{abs}\, \epsilon; c)\, A$.

The structural operational semantics (SOS) of utcc is shown in Table 1 [16]. It is given in terms of two transition relations between configurations. A configuration is of the form $\langle A, c\rangle$, where A is a process and c a store. The *internal* transition $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ states that the process A with current store c reduces, in one internal step, to the process A' with store c'. The *observable* transition $A \stackrel{(c,d)}{\Longrightarrow} B$ says that the process A on input c reduces, in one time interval, to the process B and output d. It is obtained from finite sequences of internal transitions. The symbol $\not\longrightarrow$ indicates that no internal rule can be applied.

The function $F(B)$ is used to define the observable transition. It determines the utcc process to be executed in the following time instant looking to the next and unless processes occurring in A. In particular, $F(\mathsf{skip}) = \mathsf{skip}$, $F((\mathsf{abs}\, \vec{x}; c)\, Q) = \mathsf{skip}$, $F(P_1||P_2) = F(P_1)||F(P_2)$, $F((\mathsf{local}\ x; c)\, Q) = (\mathsf{local}\ x)\, F(Q)$, $F(\mathsf{next}\, Q) = Q$ and $F(\mathsf{unless}\, c\, \mathsf{next}\, Q) = Q$. The equivalence relation $\equiv$ states when two configurations are equivalent (for instance $A||B \equiv B||A$).[1]

$\mathbf{R}_T$	$\dfrac{}{\langle \mathsf{tell}(c), d\rangle \longrightarrow \langle \mathsf{skip}, d \wedge c\rangle}$
$\mathbf{R}_P$	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', d\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, d\rangle}$
$\mathbf{R}_L$	$\dfrac{\langle A, c \wedge (\exists \vec{x}\, d)\rangle \longrightarrow \langle A', c' \wedge (\exists \vec{x}\, d)\rangle}{\langle (\mathsf{local}\, \vec{x}; c)\, A, d\rangle \longrightarrow \langle (\mathsf{local}\, \vec{x}; c')\, A', d \wedge \exists \vec{x}\, c'\rangle}$
$\mathbf{R}_U$	$\dfrac{d \vdash c}{\langle \mathsf{unless}\, c\, \mathsf{next}\, A, d\rangle \longrightarrow \langle \mathsf{skip}, d\rangle}$
$\mathbf{R}_R$	$\dfrac{}{\langle !A, d\rangle \longrightarrow \langle A \parallel \mathsf{next}\, !A, d\rangle}$
$\mathbf{R}_A$	$\dfrac{d \vdash c[\vec{y}\backslash\vec{x}] \quad \lvert\vec{y}\rvert = \lvert\vec{x}\rvert}{\langle (\mathsf{abs}\, \vec{x}; c)\, A, d\rangle \longrightarrow \langle A[\vec{y}\backslash\vec{x}] \parallel (\mathsf{abs}\, \vec{x}; c \wedge \vec{x} \neq \vec{y})\, A, d\rangle}$
$\mathbf{R}_S$	$\dfrac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}$ if $\gamma_1 \equiv \gamma_1'$ and $\gamma_2 \equiv \gamma_2'$
$\mathbf{R}_O$	$\dfrac{\langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow}{A \stackrel{(c,d)}{\Longrightarrow} F(B)}$

Table 1. Internal and observable reductions of utcc.

[1] See [16] for the complete definitions.

2.3 Timed Concurrent Constraint Language

The Timed Concurrent Constraint Language (tccp in short) is a declarative language introduced in [3] to model concurrent and reactive systems. The language inherits all the features of the cc paradigm, including the monotonicity of the store. An operator for dealing with negative information and an implicit notion of time are newly defined. Regarding the notion of time, instead of having time intervals, in tccp each update or consult to the store takes one time instant. Different consults and/or updates in parallel take one single time instant. Let us briefly recall the syntax of tccp [3]. A tccp program P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(\vec{x}) :- A$, and A is an agent:

$$A, B, A_i ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B \mid A \parallel B \mid \exists x\, A \mid \mathsf{p}(\vec{x})$$

where c, c_i are *finite constraints* of $\mathcal{C}$. Similarly to utcc, the skip agent does nothing and $\mathsf{tell}(c)$ adds the constraint c to the store. tccp is non-deterministic, thus the choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ executes (in the following time instant) one of the agents A_i provided its guard c_i is satisfied. In case no condition c_i is entailed, the choice agent *suspends* (and it is again executed in the following time instant); The conditional agent now c then A else B executes agent A if the store satisfies c, otherwise executes B; $A \parallel B$ executes the two agents A and B in parallel (following the *maximal parallelism* model); The $\exists x\ A$ agent is used to define the variable x local to the process A. We use $\exists \vec{x}\ A$ to represent $\exists x_1 \ldots \exists x_n\ A$; Finally, $\mathsf{p}(\vec{x})$ is the procedure call agent where $\vec{x}$ denotes the set of parameters of the process p. Regarding time passing, only the tell, choice and procedure call agents consume time.

As in the original model of cc, and differently from utcc, the store behaves monotonically, thus it is not possible to change the value of a given variable along the time. Similarly to the logic approach, to model the evolution of variable values along the time we can use the notion of *stream*. For instance, given an *imperative-style* variable, we write $X = [Y \mid Z]$ to denote a stream X recording the current value Y, and the future values in the stream Z. A specific function $\mathsf{find}(X, T)$ is defined on the constraint system to consult whether a term can be retrieved from a stream in the store.

Definition 2 (Find). *Let X be a stream, I a constraint and st the current store at time instant t. Then, $\mathsf{find}(X, I)$ returns true if I can be retrieved from the stream X; Otherwise returns false. Formally,*

$$\mathsf{find}(X, I) = \begin{cases} \mathit{true} & \mathit{if}\ st \vdash_t X \dot{\supset} I \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

where $\dot{\supset}$ denotes the notion of term retrieved from a stream, i.e., it checks if I unifies with, at least, one of the values stored in the given stream. For example, a call to $find$([a(1), b(2) | T],a(Ag)) would return true provided the variable Ag is non-instantiated in the store.

In this paper, we follow the modified computation model for tccp presented in [1] where the store was replaced by a *structured store* in order to ease the task of updating and retrieving information from the store. A *structured store* [1] consists of a timed sequence of stores st_i where each store contains the information added at the i-th time instant. Table 2 shows the operational semantics of tccp in terms of a transition system over configurations. Each transition step takes one time unit. The configuration $\langle A, st\rangle$ represents the agent A to be executed under the store st.

R1	$\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} c\rangle_{t+1}$	
R2	$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$	if $0 \leqslant j \leqslant n, st \vdash_t c_j$
R3	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$	if $st \vdash_t c$
R4	$\dfrac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$	if $st \nvdash_t c$
R5	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$	if $st \vdash_t c$
R6	$\dfrac{\langle B, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$	if $st \nvdash_t c$
R7	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B', st' \sqcup st''\rangle_{t+1}}$	
R8	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \not\longrightarrow}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B, st'\rangle_{t+1}}$	
R9	$\dfrac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x\, A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x\, A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$	
R10	$\langle \mathsf{p}(\vec{x}), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$ if $\mathsf{p}(\vec{x}) : -A \in D$	

Table 2. Operational semantics of the tccp language.

Rule **R1** describes the behavior of the tell agent at time instant t, which augments the store st by adding the constraint c. The constraint c will be available to other agents from the time instant $t+1$. Rule **R2** states that A_j is executed in the following time unit whenever st entails the condition c_j. Regarding the conditional agent, **R3** models the case in which the agent A with the current store st is able to evolve into the agent A' and a new store st' and st satisfies c, then A' is executed in the following time instant with the computed store st'. **R7** models the evolution of the parallel agent: if A with store st is able to evolve into A' with a new computed store st', and also B with store st is able to evolve into an agent B' with st'', then $A'||B'$ is run in the following time instant with the store resulting from the conjunction of st' and st''. The rest of rules are interpreted in a similar way.

3 Embedding utcc into tccp

In this section, we show how utcc can be embedded into tccp. It is not possible to have a direct transformation since the two languages have important differences in nature. In particular, both languages handle time differently, and the abstraction operator of utcc is not present in tccp. We define a transformation which, given the specification of an utcc program, constructs in an automatic way a tccp program that preserves the expected behavior of the original utcc program. Let us first show the intuition of the transformation by means of an example. The following example is extracted from [16], where it was used to illustrate how mobility can be modeled in utcc:

Example 1 (Mobility [16]). Let Σ be a signature with the unary predicates out_1, out_2, ... and a constant 0. Let Δ be the set of axioms over Σ valid in first-order logic. Consider the following process specifications.

$P = (\mathsf{abs}\, y; out_1(y))\, \mathsf{tell}(out_2(y))$

and

$Q = (\mathsf{local}\, z)(\mathsf{tell}(out_1(z)) \| \mathsf{when}\, out_2(z)\, \mathsf{do\, next\, tell}(out_2(0))).$

The process models a method to detect when a private information can become public in a given (non secure) channel (out_2). Thus, we have two channels called out_1 and out_2. Process P recovers from the first channel a value and forwards it to the second one. Q sends a private value (locally declared) z to out_1 and, in parallel, checks whether the second channel receives such value. In case the value appears in the out_2 channel, then, in the following time interval, the value 0 is sent to the second channel (modeling the vulnerability detection). By executing both processes in parallel $P \| Q$, the value 0 is produced in the second time interval.

Our transformation generates as many tccp procedure declarations as utcc processes are defined. In this case, we get the procedure declarations `proc_P` and `proc_Q`. Then, the initial call $P \| Q$ is transformed into the initial call (agent) of the tccp program.

We start by showing the transformation for the Q process. In this case, we have to pay special attention to model the utcc timed operator next, since it does not exist in tccp. The utcc process states that, when z is retrieved in the second channel, then in the following time interval (after the resting point) the value 0 is emitted to the channel. We define a global stream *Syn* that models when the utcc computation reaches a resting point. Having this in mind, we emit the 0 value when an `ok` is the value of the stream. At this point of the paper we omit the details for the computation of the synchronization mechanism (the values for the synchronization stream), but we present it in the next section.

$$\texttt{proc\_Q}() \text{ :- } \exists\, z\, (\mathsf{tell}(out_1(z)) \| \mathsf{ask}(out_2(z)) \rightarrow \mathsf{ask}(Syn \doteq \texttt{ok}) \rightarrow \mathsf{tell}(out_2(0))).$$

The transformation for the P process is a bit more elaborated due to the use of the abs operator. Remember that (abs y; $out_1(y)$) tell($out_2(y)$) executes (in parallel) the different instantiations (depending on the possible replacements of y) of the tell agent. In order to handle the different possible instantiations, we use an auxiliary declaration $\texttt{abs}_i$ that simulates the behavior of the abs operator and a term $\texttt{subst}_\texttt{i} \in \Sigma$ to accumulatively store the replacement found in each iteration.

$$\texttt{proc\_P}() \text{ :- } \exists\, y, S\,(\texttt{abs}_1(y, S) \parallel \mathsf{tell}(\texttt{subst}_1(S))).$$

$$\begin{aligned}\texttt{abs}_1(y, S) \text{ :- } \exists\, t, S'\,((&\mathsf{now}(out_1(y)\{t\backslash y\} \wedge \neg(\mathsf{find}(S, \{t\backslash y\}))) \\ &\mathsf{then}\ (\mathsf{tell}(out_2(y)\{t\backslash y\}) \parallel \\ &\quad (\mathsf{tell}(S = [\{t\backslash y\} \mid S']) \parallel \\ &\quad\ \texttt{abs}_1(y, S))) \\ &\mathsf{else}\ \mathsf{skip})).\end{aligned}$$

The arguments in the auxiliary call determine the pattern that is defined in the conditional agent. After executing the agent $\mathsf{tell}(S = [\{t\backslash y\} \mid S'])$ the stream S includes the replacement of y handled in the current recursive call. The computation stops when there is no replacement retrieved from the store that is not already in the stream S. The last part of the example is the translation of the utcc initial process $P \| Q$:

$$\begin{aligned}&(\texttt{proc\_P} \parallel \texttt{proc\_Q} \parallel \mathsf{tell}(Syn = [\texttt{wait} \mid C_T]) \\ &\ \mathsf{utcc\_clock}(out_1(z) \wedge out_2(z), Syn) \parallel\ \mathsf{utcc\_clock}(out_2(0), Syn))\end{aligned}$$

The two processes are called in parallel, as in the utcc example, and also the synchronization stream is initialized to the value wait, since the computation has not reached a resting point. Actually, also two clocks (one for each utcc time unit) are defined and executed in parallel to the above specification. Each clock updates the value of Syn whenever a condition (the first parameter of the call which characterizes a utcc resting point) holds. The conditions are statically computed, and we show the formalization in the following section. Here we simply show how we simulate the pass of the time of utcc in tccp by using a procedure call agent. The declaration for such procedure call utcc_clock is shown below. It contains two parameters, the first one represents the information computed in a given utcc time unit, whereas the second one is the synchronization variable Syn as we explain above. Each time a resting point is reached (the condition of the choice agent, identified by the variable $Store_i$, holds), the synchronization stream is updated to ok, thus all the agents that were waiting for this value would start their execution. Then, the value is set again to wait.

$$\begin{aligned}&\texttt{utcc\_clock}(Store_i, Syn) \text{ :-} \\ &\quad \mathsf{ask}(Store_i) \rightarrow \exists\, Sn, Sn_1((\mathsf{tell}(Syn = [\texttt{ok} \mid Sn) \parallel \\ &\qquad\qquad \mathsf{ask}(true) \rightarrow \mathsf{tell}(Sn = [\texttt{wait} \mid Sn_1]))).\end{aligned}$$

In Table 3, we show the trace of the execution of the resulting tccp program. The table is interpreted from left to right, up to bottom. It is shown for each

(tccp) time instant the current store and the agent to be executed. In the original utcc example, at the second resting point the value $out_2(0)$ is emmited. In the tccp trace, it can be seen that only after the first $Syn \doteq \texttt{ok}$ holds (thus the first resting point has been reached), the given value is emmited to the store. The value of the store at time instant 6 means the same as $Syn \doteq \texttt{ok}$. We have simplified notation at instant 4 to improve clarity. Really, the constraint $C_T = [\texttt{ok} \mid Sn]$, which implies $Syn \doteq \texttt{ok}$, is the information available in the store at time instant 4.

time	0	1
store	true	$Syn = [\texttt{wait} \mid C_T]$
agents	proc_P() ‖ proc_Q() ‖ tell($Syn = [\texttt{wait} \mid C_T]$) ‖ utcc_clock($out_1(z) \wedge out_2(z)$,$Syn$)‖ utcc_clock($out_2(0)$,$Syn$)	$\mathsf{abs}_1(y, S)$ ‖ $\mathsf{tell}(\mathrm{subst}_1(S))$ tell($out_1(z)$) ‖ ask($out_2(z)$)→ … ‖ ask($out_1(z) \wedge out_2(z)$)→ … ‖ ask($out_2(0)$)→ …
time	2	3
store	$out_1(z)$, $\mathrm{subst}_1(S)$	$t = z$, $out_2(z)$, $S = [\{\{t \backslash y\}\} \mid S']$
agents	now($out_1(y)\{t\backslash y\}\wedge$ ¬(find($S, \{\{t\backslash y\}\}$)))… tell($out_2(y)\{t\backslash y\}$) ‖ tell($S = [\{t\backslash y\} \mid S']$) ‖ abs($y, S$) ‖ ask($out_2(z)$)→ … ‖ ask($out_1(z) \wedge out_2(z)$)→ … ‖ ask($out_2(0)$)→ …	now($out_1(y)\{t'\backslash y\}\wedge$ ¬(find($S, \{\{t'\backslash y\}\}$)))… skip ‖ ask($Syn \doteq \texttt{ok}$) → tell($out_2(0)$) ‖ $\exists\, Sn, Sn_1$(tell($Syn = [\texttt{ok} \mid Sn]$) ‖ ask($true$)→ tell($Sn = [\texttt{wait} \mid Sn_1]$))‖ ask($out_2(0)$)→ …
time	4	5
store	$Syn \doteq \texttt{ok}$	$out_2(0)$, $Sn = [\texttt{wait} \mid Sn_1]$
agents	tell($Sn = [\texttt{wait} \mid Sn_1]$)‖ ask($out_2(0)$)→ …	$\exists\, Sn', Sn'_1$ (tell($Syn = [\texttt{ok} \mid Sn']$) ‖ ask($true$)→ tell($Sn' = [\texttt{wait} \mid Sn'_1]$))
time	6	7
store	$Sn_1 = [\texttt{ok} \mid Sn']$	$Sn' = [\texttt{wait} \mid Sn'_1]$
agents	tell($Sn' = [\texttt{wait} \mid Sn'_1]$)	

Table 3. Trace of the resulting tccp program.

4 Formalization of the transformation

The transformation from utcc into tccp can be divided in two phases. The first one encodes each utcc process into tccp code, whereas the second one defines the necessary synchronization among the generated tccp agents. As we have shown, since the notion of time of both languages differs, we need to force the synchronization of processes in order to mimic the behavior of the original utcc program.

In the following, we define the τ_P function that transforms the utcc program into a tccp program. As mentioned above, this function includes a final phase where the synchronization mechanism complements the first transformation phase. Next we show the formalization. In [10] we show in pseudocode the mechanization of the process. We also prove that the built tccp program mimics the behavior of the given utcc program.

We say that an utcc program, similarly to a tccp program, is composed by a set of utcc declarations of processes and the utcc process that starts the execution of the program. Let us first introduce some notation. $\mathtt{u}_d$ is a declaration from the set of utcc declarations, and $\mathtt{u}_r$ is an utcc process. Moreover, we say that $\mathsf{name}(\mathtt{u}_d)$ recovers the declaration name, and $\mathsf{body}(\mathtt{u}_d)$ recovers the process on the rhs of the declaration. For example, $\mathsf{name}(\mathtt{P = tell}(out_2(y)))$ returns P.

For each declaration $\mathtt{u}_d$ in the set of declarations of the original utcc program, we define a tccp declaration in the resulting tccp program. That tccp declaration has the form:

$$\mathsf{name}(\mathtt{u}_d) \ \mathtt{:-} \ \ \tau_A(\mathsf{body}(\mathtt{u}_d)).$$

where τ_A is an auxiliary function that, given the utcc process $\mathtt{u}_r$ ($\mathtt{u}_r = \mathsf{body}(\mathtt{u}_d)$), constructs a tccp agent that mimics its behavior. Let us now describe the τ_A function. Depending on the form of the input process $\mathtt{u}_r$, τ_A behaves differently. There are nine possible cases:

Case $\mathtt{u}_r \equiv$ skip. The corresponding tccp agent is skip.[2]

Case $\mathtt{u}_r \equiv$ tell(c). The corresponding tccp agent is tell(c).

Case $\mathtt{u}_r \equiv$ (local $\vec{x}; c$) A. The corresponding tccp agent is $\exists^c \vec{x}\,(\tau_A(A))$
The superscript c denotes the initial store in the local computation in A, in the same sense as it is used in the utcc semantics for the local operator.

Case $\mathtt{u}_r \equiv A \parallel B$. The corresponding tccp agent is $(\tau_A(A) \parallel \tau_A(B))$.

Case $\mathtt{u}_r \equiv$ next A. The corresponding tccp agent is $\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow \tau_A(A)$.
The Syn variable is a synchronization stream that is updated depending on the clock of the utcc program. The symbol $\doteq$ checks the last (current) value of the stream, which is updated to ok each time a resting point is reached in the utcc program. We introduce later the synchronization mechanism that takes care of updating Syn.

Case $\mathtt{u}_r \equiv$ unless c next A. The corresponding tccp agent is
$\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow$ now c then skip else $\tau_A(A)$.

Case $\mathtt{u}_r \equiv\ !A$. The corresponding tccp agent is $(\tau_A(A) \parallel \mathtt{aux}_i)$
$\mathtt{aux}_i$ is an auxiliary (fresh) declaration defined to simulate the replication of utcc by means of the recursion capability of tccp. The definition of the new declaration is: $\mathtt{aux}_i$:- $\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow\ (\tau_A(A) \parallel \mathtt{aux}_i)$, where at each resting point ($Syn \doteq \mathtt{ok}$ holds) the execution of $\tau_A(A)$ in parallel with the procedure call modeling recursion are executed.

[2] Since the semantics of both the utcc version of the agent and the tccp version of the agent behave similarly and no confusion can arise, we don't distinguish them syntactically.

Case $\mathtt{u}_r \equiv (\mathsf{abs}\,\vec{x}; c)\ A$**.** The corresponding tccp agent is $\exists\,\vec{x}, S\,(\mathtt{abs}_i(\vec{x}, S) \parallel \mathsf{tell}(\mathtt{subst}_i(S)))$, where $\mathtt{abs}_i$ is an auxiliary (fresh) tccp declaration defined as follows:

$$\begin{aligned}\mathtt{abs}_i(\vec{x}, S) :\!\!- \exists\,\vec{t}, S'\,(&\mathsf{now}\,(c\{\vec{t}\backslash\vec{x}\} \wedge \neg(\mathsf{find}(S, \{\vec{t}\backslash\vec{x}\})))\\ &\mathsf{then}\ \ (\tau_A(A\{\vec{t}\backslash\vec{x}\}) \parallel\\ &\qquad (\mathsf{tell}(S = [\{\vec{t}\backslash\vec{x}\} \mid S'])\parallel\\ &\qquad\ \mathtt{abs}_i(\vec{x}, S)))\\ &\mathsf{else}\ \mathsf{skip}).\end{aligned}$$

Where $\mathtt{subst}_i \in \Sigma$, i.e., it is a predicate handled by the constraint system $\mathcal{C}$ and identifies the stream storing the replacement (substitution) found in each iteration.

Case $\mathtt{u}_r \equiv P = A$**.** The corresponding tccp agent is the call to the process $\mathsf{name}(P)$, i.e., `proc_P`.

Once all the utcc declarations are transformed, the second phase of the transformation τ_A can start. It consists in defining the clock that mimics the time passing in utcc. This is necessary since the explicit notion of time of utcc differs from the implicit one of tccp: A time unit in utcc may correspond with several time units in tccp. We use the stream *Syn* to simulate such clock. The stream may contain the following values:

- `wait`, that means that a resting point has not been reached.
- `ok`, that simulates that the resting point of the current time interval has been reached.

As a result of this second transformation phase, a new declaration `utcc_clock` to compute the time passing of utcc in tccp is introduced. Then, the initial call of the program will contain in parallel many procedures call to `utcc_clock` (with different conditions) as necessary to simulate the utcc clock. We need to run a pre-process that *computes* each time unit of such clock, i.e., it identifies the resting points of the original utcc program that define when a tick must occur. In the following, we show the function *instant* that computes the store generated during one utcc time instant. This information is used by the clock `utcc_clock` to update the synchronization variable *Syn*. The auxiliary function *follows* is used to compute the process that must be executed in the following time instant, and simulates the partial function F of the utcc semantics.

Given the utcc program $\mathtt{u}_p$, let us assume that there are n process declarations in the program. First, we define a tccp declarations of the following form.

$$\begin{aligned}&\mathtt{utcc\_clock}(\mathit{Store}_i, \mathit{Syn})\ :\!\!-\\ &\quad \mathsf{ask}(\mathit{Store}_i) \rightarrow \exists\,\mathit{Sn}, \mathit{Sn}_1\,((\mathsf{tell}(\mathit{Syn} = [\mathtt{ok} \mid \mathit{Sn}])\ \parallel\\ &\qquad\qquad\qquad \mathsf{ask}(\mathit{true}) \rightarrow\ \mathsf{tell}(\mathit{Sn} = [\mathtt{wait} \mid \mathit{Sn}_1]))).\end{aligned}$$

Then, for each store computed by the functions *instant* given the process computed by *follows*, we specify in the initial term, in parallel, a call to such declaration with the computed store. This means that we define n runs for the

above declaration. When new computed stores have already been generated (modulo renaming), then the process ends. More specifically, the computation starts by computing the *instant* for the initial call. Then, the function *follows* computes the process after the following resting point for which the *instant* is computed. The iteration proceeds until a *loop* is reached, i.e., the computed *instant* has already been computed (modulo renaming).

The function *instant* computes the information at the resting point of a given time interval, following the operational semantics of utcc:

$$instant(\mathtt{u}_r, st) = \begin{cases} st & \text{if } \mathtt{u}_r \equiv \mathsf{skip} \\ st \cup c & \text{if } \mathtt{u}_r \equiv \mathsf{tell}(c) \\ instant(A\{\vec{x}'\backslash\vec{x}\}, c \wedge \exists\vec{x} st) & \text{if } \mathtt{u}_r \equiv (\mathsf{local}\ \vec{x}; c)\ A \text{ and } \vec{x}' \text{ are fresh variables} \\ instant(A_1, st) \wedge instant(A_2, st) & \text{if } \mathtt{u}_r \equiv A_1 \parallel A_2 \\ st & \text{if } \mathtt{u}_r \equiv \mathsf{next}\ A \\ st & \text{if } \mathtt{u}_r \equiv \mathsf{unless}\, c\, \mathsf{next}\, A \\ instant(A, st) & \text{if } \mathtt{u}_r \equiv\ !A \\ \bigwedge_{z\in\theta} instant(A\{\vec{y}\backslash\vec{x}\}, st) & \text{if } \mathtt{u}_r \equiv (\mathsf{abs}\, \vec{x}; c)\, A \text{ and } st \vdash c\{\vec{y}\backslash\vec{x}\} \text{ and } \theta = \{\vec{y} \mid st \vdash c\{\vec{y}\backslash\vec{x}\}\} \\ instant(\mathsf{body}(\mathtt{P}), st) & \text{if } \mathtt{u}_r \equiv\ P = A \\ st & default \end{cases}$$

The function *follows* is similar to the future function F of utcc:

$$follows(\mathtt{u}_r, st) = \begin{cases} \mathsf{skip} & \text{if } \mathtt{u}_r \equiv \mathsf{skip} \\ \mathsf{skip} & \text{if } \mathtt{u}_r \equiv \mathsf{tell}(c) \\ (\mathsf{local}\ \vec{x})\ follows(A, st \wedge c) & \text{if } \mathtt{u}_r \equiv (\mathsf{local}\ \vec{x}; c)\ A \\ follows(A_1, st) \parallel follows(A_2, st) & \text{if } \mathtt{u}_r \equiv A_1 \parallel A_2 \\ A & \text{if } \mathtt{u}_r \equiv \mathsf{next}\ A \\ A & \text{if } \mathtt{u}_r \equiv \mathsf{unless}\, c\, \mathsf{next}\, A \text{ and } st \not\vdash c \\ !A & \text{if } \mathtt{u}_r \equiv\ !A \\ (\mathsf{abs}\ \vec{x}; c)\ follows(A, st \wedge c) & \text{if } \mathtt{u}_r \equiv\ (\mathsf{abs}\ \vec{x}; c)\ A \\ follows(\mathsf{body}(\mathtt{P}), st) & \text{if } \mathtt{u}_r \equiv\ P = A \\ \mathsf{skip} & default \end{cases}$$

To guarantee the termination of the process is necessary to define a notion of equivalence between utcc system states. To achieve this goal, we follow the idea of the notion of equivalence among states proposed in [5]. An state of the system is composed by a store and the process to be executed in the following time instant. The notion of equivalence defines that two states are equivalent if there exists a renaming of variables that makes the stores of both states equivalents (looking at the current value of streams) and if the processes to be executed

in the following time instant of both states coincide. Thus, each time that we compute a given store by using the function *instant*, and the processes to be executed in the following time instant by using the function *follows*, we compare the corresponding state with all the previous generated. Then, in case that we found an equivalent state we finish the computation of the clock. Due to the monotonicity of the utcc language a fixed point is always reached. We formalize the respective proofs in [10]. Moreover, the termination of the abs operator is proven in [16] by using a symbolic semantic avoiding infinite subsitutions.

4.1 Correctness of the transformation

We have defined a transformation from utcc programs into tccp programs in such a way that the resulting tccp program mimics the behavior of the original utcc program. In this section we show that our method is sound in the sense that each utcc trace of a program can be simulated by a tccp trace of the transformed version.

We first introduce some notations. Similarly to [15], we say that elements $c_1, c_2, c'_1, c'_2, \ldots$ are the set of constraints defined by $\mathcal{C}$. Then, $\alpha, \alpha', \ldots$ denote infinite sequences of constraints where $\alpha = c_1.c_2 \ldots$ and $\alpha' = c'_1.c'_2 \ldots$. Finally, $\alpha(i)$ denotes the *i-th* element in α, for instance, $\alpha(2) = c_2$. Note that any constraint by itself is (or can represent) a store, so we can use both terminologies indistinctly.

Definition 3 (Input-Output Relations over utcc processes [15]). *Given the utcc process P, $\alpha = c_1.c_2 \ldots$ and $\alpha' = c'_1.c'_2 \ldots$, $P \overset{(\alpha,\alpha')}{\Longrightarrow}$ is used to represent $P = P_1 \overset{(c_1,c'_1)}{\Longrightarrow} P_2 \overset{(c_2,c'_2)}{\Longrightarrow} \ldots$. Then, the set*

$$io_{utcc}(P) = \{(\alpha, \alpha') \mid P \overset{(\alpha,\alpha')}{\Longrightarrow}\}$$

denotes the input-output behavior of P. This sequence can be interpreted as an interaction between the system P and an environment. At the time instant i, the environment provides an input c_i and P_i produces the output c'_i.

Now we recall the definition of the observable behavior of tccp agents, based on the transition system described in Table 2. Let $d = d_0 \cdot d_1 \cdot \ldots \cdot d_i \cdot \ldots$ be a structured store where d_n is the information computed at time instant n.

Definition 4 (Observable Behavior of tccp programs [1]). *Given a tccp program P, an agent A_0, and an initial structured store $d = d_0^0 \cdot true^\omega$ where d_0^0 represents the first component of d_0, the* timed operational semantics *of P w.r.t. the initial configuration $\langle A_0, d_0 \rangle$ is defined as:*

$$io_{tccp}(P)[\langle A_0, d_0 \rangle] = \{d = d_0 \cdot d_1 \cdot \ldots \mid \langle A_i, d_i \rangle_i \longrightarrow \langle A_{i+1}, d_{i+1} \rangle_{i+1} \text{ for } i \geq 0\}$$

Let us next define the notion of inclusion of a utcc trace into a tccp trace. We write $\sqcup_{0 \leqslant k \leqslant j} d_k$ with $k, j \in \mathbb{N}$ for the conjunction of all the stores 0 to j of the given structured store d.

Definition 5 (Entailment Relation). *Given the sequence of constraints* $\alpha' = c'_1.c'_2 \ldots$ *and the structured store* $d = d_0 \cdot d_1 \cdot d_2 \cdot \ldots$ *representing the outputs of a given* utcc *program and a given* tccp *program, respectively. Let* $i, j \in \mathbb{N}$ *denoting the time instants of* utcc *and* tccp*, respectively . We say that* d *entails the sequence of constraints* α'*, denoted as* $d \vdash_\tau \alpha'$*, iff*

$$\forall i (d \vdash_j \alpha'(i)) \ s.t. \ j + 1 \geqslant i, i > 0$$

and the indexes j *are pairwise distinct.*

The following example illustrates the definition.

Example 2. Given the sequence $\alpha' = (v = 1 \wedge w = 2).z > 8.(x = 7 \wedge y = 5)$, and the structured store $d = true \cdot (v = 1) \cdot (w = 2) \cdot z > 10 \cdot x = 7 \cdot r = 3 \cdot (y = 5 \wedge q < 3)$. We can see that:

- $\sqcup_{0 \leqslant k \leqslant 2} d_k \vdash \alpha'(1)$ since it holds that $(true \wedge v = 1 \wedge w = 2) \vdash (v = 1 \wedge w = 2)$
- $\sqcup_{0 \leqslant k \leqslant 3} d_k \vdash \alpha'(2)$ since it holds that $(true \wedge v = 1 \wedge w = 2 \wedge z > 10) \vdash z > 8$
- $\sqcup_{0 \leqslant k \leqslant 6} d_k \vdash \alpha'(3)$ since it holds that $(true \wedge v = 1 \wedge w = 2 \wedge z > 10 \wedge x = 7 \wedge r = 3 \wedge y = 5 \ \wedge q < 3) \vdash (x = 7 \wedge y = 5)$

Therefore, $d \vdash_\tau \alpha'$.

Now we are ready to define the notion of trace inclusion, i.e., when a utcc trace is mimicked by a tccp trace.

Definition 6. *Given* (α,α') *the sequences of constraints computed by a* utcc *program, and* d *the structured store computed by a* tccp *program. Then,* (α,α') *is included in* d*, written* $(\alpha, \alpha') \sim_\tau d$*, iff* $d \vdash_\tau \alpha'$.

To guarantee the correctness of our method, we shall prove the correctness of the synchronization and transformation processes. All proofs of our results can be found in [10].

Lemma 1 (Correctness of Synchronization). *Consider a* utcc *program* U *of the form* $U := U_D.U_R$ *where* U_D *is the declaration set and* U_R *is the process that initiates the execution of* U*. Let* st^i *the store computed by the* i*th iteration of the function instant on the program* U*. Let* $U_R^1 = follows(U_R, st^0)$*, where* st^0 *is the initial store and* $U_R^i = follows(U_R^{i-1}, st^{i-1})$*. Let* $io_{utcc}(U) = (\alpha, \alpha')$*. Then,* $st^1 \vdash \alpha'(1)$*,* $st^2 \vdash \alpha'(2)$*, ...,* $st^n \vdash \alpha'(n)$ *s.t.:*

- $st^1 = instant(U_R^1, true)$ *and,*
- $st^n = instant(U_R^n, true)$ *s.t.* $n > 1$

The following theorem states that a given utcc trace is included in the trace generated by the equivalent tccp program, i.e., by the tccp program obtained by the transformation.

Theorem 1 (Correctness of Transformation). *Consider an* utcc *program* U*. Let* T *the* tccp *program resulting of transforming* U*,* $\tau_P(U) = T$*. Given* $io_{utcc}(U) = (\alpha, \alpha')$*, and* $io_{tccp}(T) = d$*. Then,* (α,α') $\sim_\tau d$.

5 Conclusion

This paper presents a sound transformation from utcc processes into tccp specifications. The two languages belong to the concurrent constraint paradigm, but they have very different features which make the transformation difficult. The transformation shows that tccp is expressive enough to model utcc processes. In particular, by means of a synchronization mechanism based on a shared stream that acts as a clock, the explicit notion of time of utcc can be simulated. Moreover, the new abstraction operation can be expressed in terms of parameters passing among calls. The transformation can be automatized, which would allow us to reuse the tools defined for the tccp language, such as the recent tccp interpreter.

As future work, we plan to implement the transformation and to study the relation of the language with other concurrent languages such as Linda.

References

1. Alpuente, M., Gallardo, M.M., Pimentel, E., Villanueva, A.: Verifying Real-Time Properties of tccp Programs. Journal of Universal Computer Science **12**(11) (2006) 1551–1573
2. Alpuente, M., Gallardo, M., Pimentel, E., Villanueva, A.: A semantic framework for the abstract model checking of tccp programs. Theoretical Computer Science **346**(1) (2005) 58–95
3. Boer, F.S.d., Gabbrielli, M., Meo, M.C.: A Timed Concurrent Constraint Language. Information and Computation **161**(1) (2000) 45–83
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1987) 178–188
5. Falaschi, M., Villanueva, A.: Automatic Verification of Timed Concurrent Constraint Programs. Theory and Practice of Logic Programming **6**(3) (May 2006) 265–300
6. Gupta, V., Saraswat, V.A., Jagadeesan, R.: Foundations of Timed Concurrent Constraint Programming. In: Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science, Paris, France, IEEE Computer Society Press (July 1994) 71–80
7. Harel, D., Pnueli, A.: On the development of reactive systems. (1985) 477–498
8. Lescaylle, A., Villanueva, A.: Using tccp for the Specification and Verification of Communication Protocols. In: Proceedings of the 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07). (2007)
9. Lescaylle, A., Villanueva, A.: Verification and Simulation of protocols in the declarative paradigm. Technical report, DSIC, Univeridad Politécnica de Valencia (2008) Available at http://www.dsic.upv.es/~alescaylle/files/dea-08.pdf.
10. Lescaylle, A., Villanueva, A.: Bridging the gap between two Concurrent Constraint Languages. Technical report, DSIC, Univeridad Politécnica de Valencia (2009) Available at http://www.dsic.upv.es/~villanue/LV09-techrep.pdf.
11. Lescaylle, A., Villanueva, A.: The tccp Interpreter. Electron. Notes Theor. Comput. Sci. **258**(1) (2009) 63–77
12. Maraninchi, F.: Operational and compositional semantics of synchronous automaton compositions. In: In CONCUR. LNCS 630, Springer-Verlag (1992) 550–564

13. Nielsen, M., Palamidessi, C., F.D., V.: Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. Nordic Journal of Computing **9**(1) (2002) 145–188
14. Nielsen, M., Palamidessi, C., Valencia, F.D.: On the expressive power of temporal concurrent constraint programming languages. In: PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2002) 156–167
15. Olarte, C., Valencia, F.D.: The expressivity of universal timed CCP: undecidability of Monadic FLTL and closure operators for security. In: PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2008) 8–19
16. Olarte, C., Valencia, F.D.: Universal concurrent constraint programing: symbolic semantics and applications to security. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, New York, NY, USA, ACM (2008) 145–150
17. Saraswat, V.A.: Concurrent Constraint Programming Languages. PhD thesis, Carnegie-Mellon University, Cambridge, MA (January 1989)
18. Saraswat, V.A.: Concurrent Constraint Programming Languages. The MIT Press, Cambridge, MA (1993)
19. Tini, S.: On the Expressiveness of Timed Concurrent Constraint Programming. In: Electronics Notes in Theoretical Computer Science, Electronics (1999)

Large Scale Random Testing with QuickCheck on MapReduce Framework

Shigeru Kusakabe, Yuuki Ikuta

Grad. School of Information Science and Electrical Engineering, Kyushu University
744, Motooka, Nishi-ku, Fukuoka city, 819-0395, Japan

Abstract. Testing plays an important role in gaining confidence for quality, robustness, and correctness of software. Among many tools for testing, a property-based testing tool, QuickCheck, supports a high-level approach to testing Haskell programs by automatically generating random input data. Users of QuickCheck can customize their test case generation. Since increasing the number of tests is effective in obtaining higher confidence, huge number of tests may be performed especially in developing mission-critical software. We can run such kind of testing on powerful platforms such as PC-clusters in developing highly reliable software. We consider an approach to leveraging the power of testing by using "Cloud" to perform arbitrary scale random testing with QuickCheck. We employ Hadoop framework, an implementation of MapReduce programming model, which is a programming model for large-scale data-parallel applications. We can automatically distribute the generation of test data and the execution of tests in a scalable data-parallel manner.

1 Introduction

Functional programming has many advantages including features useful to create short, fast, and safe software[1]. One of the features in using a pure functional language, such as Haskell, is making us think software at a higher level, and programs serve like a kind of executable specification in formal methods. We can also translate a large part of the formal model into programs in a functional programming language as formal specification languages share several features with functional programming languages. There have been works focusing on their relations[2][3].

In light-weight formal methods, we do not rely on very rigorous means such as theorem proofs, while various formal methods are useful in developing highly reliable mission-critical software. Instead, in order to increase confidence in our specifications, we use adequately less rigorous means, such as testing executable specifications. While the specific level of rigor depends on the aim of the project, millions of tests may be conducted in developing highly reliable mission-critical software in a light-weight formal approach. For example, in an industrial project using VDM++, a model-oriented formal specification language[4], they developed

formal specifications of 100,000 steps including test cases (about 60,000 steps) and comments written in the natural language, and they carried out about 7,000 black-box tests and 100 million random tests [5].

Even if a model of the target system is correct, its corresponding implementation code may be incorrect. One approach for gaining confidence in implementation code is to test it against an executable specification of the model. We can compare the result of the implementation for a test data with the result of the executable specification for the same test data. If the results are equal to each other, we can gain confidence for the implementation. We are trying to develop this kind of model-based testing framework for software projects using executable formal specification languages. Our goal is to perform this kind of model-based testing for as many test cases as the aim of the project requires with less human intervention.

Regarding the number of test cases, preparing an arbitrary large number of test cases by hand is possible but impractical. Among many tools for testing, a property-based testing tool, QuickCheck, supports a high-level approach toward testing Haskell programs by automatically generating random input data[6]. Users of QuickCheck can customize their test case generation including the number of test cases. Since increasing the number of tests is effective in obtaining higher confidence, millions of tests may be performed on powerful platforms such as PC-clusters or Grid computing platforms[7].

We consider an approach to leveraging the power of testing by using QuickCheck on an elastic "Cloud" platform. We can perform testing of arbitrary scale by exploiting such a combination. We employ Hadoop framework to easily execute tests in a data-parallel way. Hadoop is an implementation of MapReduce programming model, which is a programming model for large-scale data-parallel applications. We can automatically distribute the generation of test data and the execution of tests in a scalable manner with Hadoop.

The rest of this paper is organized as follows. Section 2 briefly introduces QuickCheck. Section 3 explains MapReduce programming model. Section 4 discusses our implementation issues and presents the results of preliminary performance evaluation. Section 5 concludes.

2 QuickCheck

QuickCheck is an automatic testing tool for Haskell programs. It defines a formal specification language to state properties. Properties are universally quantified over their arguments implicitly. The function `quickCheck` checks whether the properties hold for randomly generated test cases when they are passed as its arguments. QuickCheck has been widely used and inspired related studies[8][9][10]

For the explanation, we use a simple `qsort` example from a book [11].

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
    where lhs = filter  (< x) xs
```

```
        rhs = filter (>= x) xs
```

We use idempotency as an example invariant to check that the function obeys the basic rules a sort program should follow. Applying the function twice has the same result as applying it only once. This invariant can be encoded as a simple property. The QuickCheck convention in writing test properties is prefixing with `prop_` to distinguish them from normal code. This idempotency property is written as a following Haskell function. The function states equality that must hold for any input data that is sorted.

```
prop_idempotent xs = qsort (qsort xs) == qsort xs
```

QuickCheck generates input data for this `prop_idempotent` and passes it to the property via the `quickCheck` function. Following example shows the property holds for the 100 lists generated.

```
> quickCheck (prop_idempotent :: [Integer] -> Bool)
OK, passed 100 tests.
```

While the sort itself is polymorphic, we must specify a fixed type at which the property is to be tested. The type of the property itself determines which data generator is used. The `quickCheck` function checks whether the property is satisfied or not for all the test input data generated.

QuickCheck has convenient features such as quantifiers, conditionals, and test data monitors. In addition to the type-based default random test data generators, it provides an embedded language for specifying custom test data generators.

3 MapReduce Programming Framework

We consider an approach to leveraging the power of testing by using "Cloud" to perform large scale testing. Increasing the number of tests can be effective in obtaining higher confidence. We prepare our testing platform in an elastic manner and perform tests in a data parallel way for test cases generated by using and customizing QuickCheck.

3.1 Elastic platform

The Cloud Computing paradigm seems to bring a lot of changes in many IT fields. We believe it also has impact on the field of software engineering and consider an approach to leveraging light-weight formal methods by using Cloud Computing which has the following aspects[12]:

1. The illusion of infinite computing resources available on demand, thereby eliminating the need for Cloud Computing users to plan far ahead for provisioning;
2. The elimination of an up-front commitment by Cloud users, thereby allowing organizations to start small and increase hardware resources only when there is an increase in their needs; and

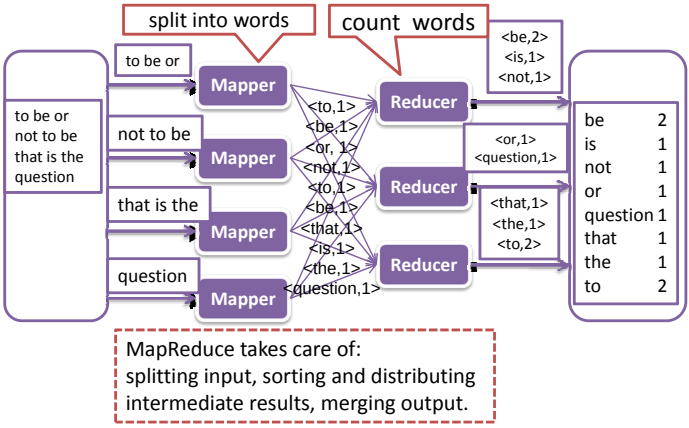

Fig. 1. Word count example in MapReduce.

3. The ability to pay for use of computing resources on a short-term basis as needed and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

We can prepare a platform of desired configuration depending on the needs of the project. In the next section, we briefly explain a software framework to exploit such platform.

3.2 MapReduce

MapReduce programming model is proposed in order for processing and generating large data sets on a cluster of machines[13]. Programs are written in a functional style, in which we specify mapper functions and reducer functions. Input data-set is split into independent chunks, and the mapper tasks process the chunks in a parallel manner. The outputs of the mappers are sorted and sent to the reducer tasks as their inputs.

MapReduce programs are automatically parallelized and executed on a large cluster of machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. Its implementation allows programmers to easily utilize the resources of a large distributed system without expert skills for parallel and distributed systems.

Fig.1 shows an outline of a simple application, WordCount, that counts the number of occurrences of each word in a given input set. A mapper processes a key/value pair to generate a set of intermediate key/value pairs. A reducer function merges all intermediate values associated with the same intermediate

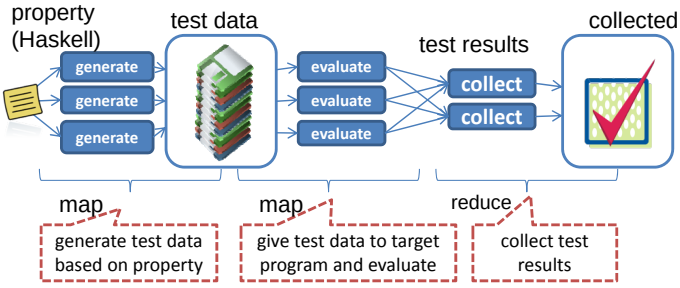

Fig. 2. Outline of our approach.

key. The application reads an input file and a mapper processes one line at a time. A key is a line number and a value is a string for a line. It then splits the line into tokens separated by whitespaces, and emits a key/value pair of <*word*, 1>. The output of each mapper is passed through the combiner for aggregation, after being sorted on the keys. The reducer sums up the values, which are the occurrence counts for each key (i.e. words in this example).

Conceptually, we can evaluate property expressions in property-based random testing in a data-parallel style by using MapReduce framework. Each mapper evaluates the property for one of the test data and reducer combines the results from mappers. By applying an automatic testing tool such as QuickCheck on MapReduce framework, we expect we can greatly reduce the cost of a large scale testing.

4 Implementation

We develop our testing environment by customizing QuickCheck on Hadoop framework. In this section, we discuss implementation issues and examine the results of our preliminary evaluation.

Our approach to implementing property-based testing on Hadoop is to separate the testing process into two phases. We generate test data by using mappers, and store the data into a file in the first phase. Then we can read and split the file, and distribute the data to mappers, where the property function written in Haskell is evaluated. Fig.2 outlines this approach.

4.1 Hadoop streaming

Hadoop, open source software written in Java, is a software framework implementing MapReduce programming model[14]. We write mapper and reducer functions in Java by default in this Hadoop framework. However, Hadoop distribution

Table 1. The ratio of unique data and distribution in generating random data of `Int` and `Float`.

		Original	Naive	Modified
`Int`	Unique(%)	34.2	8.2	34.3
	Distribution	778	97	774
`Float`	Unique(%)	99.6	95.6	99.6
	Distribution	7668	962	7707

contains a utility, Hadoop streaming, which allows us to create and run jobs with any executable or script as the mapper and/or the reducer. The utility will create a mapper/reducer job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes. When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized. This Hadoop streaming is useful in executing and testing program written in Haskell on MapReduce framework.

4.2 Redundant test data

We generate the specified number of random test data with mappers in Hadoop environment in a distributed way. However, naively splitting the number and assigning the sub-numbers to mappers lead to useless computing due to redundant test data generated in different mappers. We need to avoid increasing the number of redundant test data from the view point of efficiency and coverage. We modified the generator in QuickCheck to avoid this problem. We add one extra parameter to `check` function in QuickCheck module. The parameter represents the start index of test data and is passed to `test` function. After the total number of tests is determined, each mapper is given different start index according to the number of tests assigned to mappers. In order to see the effect of this modification, we compare the number of unique (non-redundant) test data in generating 8000 and 80000 data for `Int` and `Float` type. Table 1 shows the result. "Original" means using the QuickCheck original data generator on non-Hadoop environment. "Naive" means each data generator for subset starts its index from 1 in generating random test data on mappers in Hadoop environment. "Modified" means each data generator knows its own starting index, each of which is different from each other. According to the result, we can see effectiveness of our modification as we see no outstanding difference between "Original" and "Modified" while we have some degradation in "Naive".

4.3 Effect of distributed execution

In order to examine the effectiveness of our approach, we measured elapsed time in running QuickCheck for `prop_idempotent` on Hadoop, while changing the number

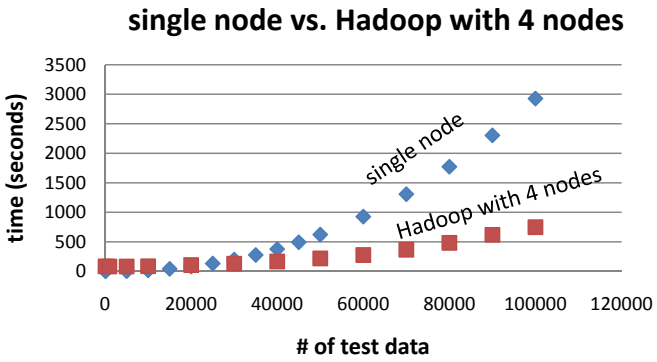

Fig. 3. Elapsed time in increasing the number of tests on a single node and on a Hadoop environment with four nodes.

of test cases. We show the result in Fig. 3. As we see in Fig. 3, until the number of tests exceeds about 30,000, the total elapsed time of Hadoop version is slower than that of non-Hadoop version while the former uses four nodes and the latter uses only a single node. The results of Hadoop version include some overheads such as distributing test data and collecting evaluation results over network. After that point, the gap between the two version becomes wider, or Hadoop version gets faster, as the number of test data increases. Thus, our approach is suitable for large scale testing.

4.4 Scalability

We evaluate the scalability of our approach when increasing the number of mapper tasks. The program used is a kind of packet analysis software developed internally. We change the number of mapper tasks for generating test data and evaluating property.

We show the result in Fig. 4. The speedup ratio is caluculated against the result of the normal QuickCheck version. As we see in Fig. 4, the increase of the number of mapper is effecive. However, the speed up ratio against the number of mapper is not ideal. While one of the reasons is overhead of using Hadoop, we will investigate further to achieve more efficient environment.

5 Concluding Remarks

We believe increasing the number of tests in a formal manner is effective in obtaining higher confidence. We considered an approach to leveraging the power

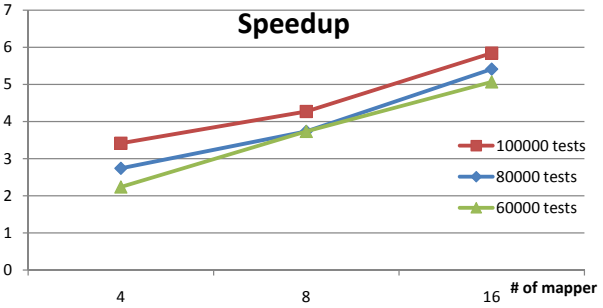

Fig. 4. Speedup ratio in chainging the number of mappers on a Hadoop environment.

of property-based testing by using QuickCheck and elastic "Cloud" to perform large scale automated testing. We used Hadoop framework, which implements MapReduce programming model, to generate test data and execute test cases in a data-parallel way. We develop a prototype and measured preliminary performance results. We generated random data as the same distribution as original QuickCheck. We also observed scalable performance for large scale testing for a small set of programs. We will further continue our work to develop more elaborate environment and show the feasibility for various kinds of software.

References

1. Hughes, J.: Why functional programming matters. Computer Journal **32**(2) (1989) 98–107
2. Borba, P., Meira, S.: From vdm specifications to functional prototypes. J. Syst. Softw. **21**(3) (June 1993) 267–278
3. Visser, J., Oliveira, J.N., Barbosa, L.S., Ferreira, J.a.F., Mendes, A.S.: Camila revival: Vdm meets haskell. In: First Overture Workshop. (2005)
4. Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M., Fitzgerald, J.: Validated Designs For Object-oriented Systems. Springer Verlag (1998)
5. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a formal specification language in the development of the "mobile felica" ic chip firmware for embedding in mobile phone. In: FM. (2008) 425–429
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. ACM SIGPLAN Notices **35**(9) (2000) 268–279
7. DUARTE, A., CIRNE FILHO, W., BRASILEIRO, F.V., MACHADO, P.D.L.: Gridunit: Software testing on the grid. In: Proceedings of the 28th ACM/IEEE International Conference on Software Engineering. Volume 28., ACM (2006) 779 – 782

8. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with quviq quickcheck. In: ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, New York, NY, USA, ACM (2006) 2–10
9. Boberg, J.: Early fault detection with model-based testing. In: Erlang Workshop. (2008) 9–20
10. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in erlang with quickcheck and pulse. In: ICFP. (2009) 149–160
11. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. Oreilly & Associates Inc (2008)
12. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Technical report, UCB/EECS-2009-28, Reliable Adaptive Distributed Systems Laboratory (February 2009)
13. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113
14. Hadoop: As of Jun.1, 09. http://hadoop.apache.org/core/

Automated Verification of Security Protocols in tccp*

Alexei Lescaylle, Alicia Villanueva

DSIC, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia (Spain),
alescaylle@dsic.upv.es, villanue@dsic.upv.es

Abstract. The Timed Concurrent Constraint language (tccp in short) is a declarative concurrent programming language designed to deal with concurrent and reactive systems. The interaction of agents executing security protocols can be specified in a compact way by taking advantage of non-determinism and the agent-based computational model. The constraint-based approach of the language, which is parametric w.r.t. an underlying *constraint system* allows us to handle partial information in an intuitive way. In brief, the computational model of tccp is based on agents generating (*telling*) and checking (*asking*) partial information (*constraints*) in a global *store*. In this work, we present a general, unified model for the specification of security protocols based on the concurrent constraint paradigm. We use the Needham-Schroeder public key authentication protocol along the paper to illustrate the approach and show how the tccp interpreter implemented in Maude can be used to verify safety properties.

1 Introduction

The *Concurrent Constraint Paradigm* (cc in short) [24] is a simple but powerful model that introduces a rich class of programming languages based on the notion of computing with partial information (constraints) instead of computing with explicit information (valuation of variables). The model is parametric w.r.t. a constraint system specifying the kinds of available constraints and how to interact with them. The computational model is based on agents, interacting asynchronously among them by adding (*telling*) or consulting (*asking*) constraints to the store (composed of a conjunction of constraints and where the information is never cancelled). Note that, by using constraints to represent states, a natural compression of the search space can be achieved.

The *Timed Concurrent Constraint language* (tccp in short) [5], extends the cc model with a notion of time that makes the language suitable for modeling reactive and embedded systems, namely systems which may have infinite computations, for example, operating systems, communication protocols, etc...

* This work has been supported by the Spanish MEC/MICINN under grant TIN2007-68093-C02-02, by the Generalitat Valenciana under grant Emergentes GV/2009/024, and by the Universidad Politécnica de Valencia, program PAID-06-07 (TACPAS).

Communication protocols, in particular security protocols, are used for secure communications between principals (participants) of the protocol. Due to the growth of internet-based applications, the protocols to securely communicate have received considerable attention since their correct behavior had to be guaranteed. Specifically, it is crucial to ensure secrecy and security of any communication action since these actions take place in a hostile environment. The *environment* models the way in which messages travel the network, but also the actions an intruder may carry out during the execution of a protocol. For modeling the environment, we follow the most popular approach for it, proposed by Dolev-Yao in [12]. In this model, the intruder can perform some malicious operations such as replay, compose, decrypt or replace messages. It is well known that the implementation of security protocols require a careful design process and justifies the use of formal models to guarantee that intruders cannot exploit their vulnerabilities.

In this work, we propose the tccp language for the formal analysis of security protocols. We take advantage of the agent-based and constraint-based models, inherited from cc, to specify the interaction between the honest participants of the protocol and the hostile environment that controls the protocol execution. In particular, concurrency and non-determinism of the language allow us to model the execution of more than one session of a protocol concurrently as well as the whole state-space of the system in a compact wat, thanks to the constrained nature. Note that the notion of store-as-constraint introduces a natural abstraction of system states. Moreover, the underlying constraint system can be extended (enriched) to specify, not only the most basic information, but also more vital actions such as nonce generation, computations, etc. The framework allows us to represent a wide number of protocols in an homogeneus way. We define how to specify the role of each participant and the environment as tccp declarations, and the properties representing the security requirements of the protocol. We can model check the specification of the system modeled by using the tccp model-checking approaches from [2,1,13,17]. In this paper, we carry out the verification process by using the tccpInterpreter system [18] (a tccp interpreter implemented in the high-performance reflective language Maude [7,8]). We show how to use the interpreter to verify in an automatic way safety properties that may allow us to guarantee the correctness of the specification of a given protocol. We also show, by using the logic introduced in [10], the properties that we want to verify and the mapping of them to Maude.

In previous works [15,16], we have proposed a similar methodology for specifying protocols. The methodology presented in this work is a refinement of the previous proposals. We have improved the specification of the protocol and the intruder model, making the model clearer, more compact and simpler. We have also extended the actions that an intruder can carry out in a protocol execution. Thanks to the new framework, we have reduced the tccp time instant in which we detected possible attacks. In particular, we have taken advantage of the the underlying constraint system to include some of the capabilities that were previously setted at the language abstraction level. This functionality has

been incorporated to the tccpInterpreter in such a way that we have provided the framework with a constraint system that includes the necessary operations. We use the Needham-Schroeder protocol as our running example to show our approach, but other protocols as those shown in [15,14] can also be modeled.

Related Work . The verification of security protocols has been widely treated, in the literature, through the use of different formalisms [6,20]. Recently, in [22] was presented the utcc language which allows the specification of mobile behaviors and to prove reachability properties. In [22], each participant is modeled as an independent process and the underlying constraint system carries out some specific operations related to communication protocols. One of the main differences between this approach and ours is that, following the Dolev-Yao model, in our case the attack is not explicitly modeled, but a generic specification for the intruder is used. This means that we are not detecting a known attack, but potentially any attack that the Dolev-Yao intruder is able to run. Moreover, although both languages belong to the same cc paradigm, they are quite different in nature: tccp is non-deterministic whereas utcc is deterministic, and utcc iteration is based on replication whereas tccp uses recursion. These differences make the specification of systems different in the two languages. In [9], a method to model security protocols in a real-time scenario is proposed. We follow the same idea for specifying a general Dolev-Yao intruder model. As in our approach, also the model-checking technique is used to verify the given protocol. In [4] a specific constraint system and some related techniques to analyze security protocols is developed. In tccp, the specification and verification process integrates the dependency on the constraint system with the agent-based model. In our case, we extend the underlying constraint system to represent the information resulting from the execution of a protocol such as the private knowledge of each principal, but the particular behavior of participants is not modeled within the constraint system. Maude and Haskell have been used to model security protocols [11,3]. The formalization process in both approaches is similar. Each step of the analyzed protocol is encoded in two rules representing the sending and receiving of a message, respectively. In both formalisms, a model-checking technique is used to explore the state space for possible attacks. The main different w.r.t. our proposal is that we present a general intruder specification that can be use to analyze different kinds of protocols with no modification.

The paper is organized as follows. An brief description of the tccp language is given in the next section. Section 3 shows the informal specification of a variant of the well-known Needham-Schroeder public key authentication protocol. Section 4 presents the formal specification of the Needham-Schroeder protocol and the hostile environment that models the actions an intruder may perform. In Section 5 we describe the results of the analysis of the given protocol. Finally, in Section 6 we discuss some conclusions and future work.

2 The tccp language

The Timed Concurrent Constraint language inherits from cc the agents defined in this paradigm and adds a new conditional agent for dealing with negative information. A tccp program P is of the form $D.A$ where D is a set of declarations of the form $\mathsf{p}(x) :- B$ and A is an agent that initiates the execution. The syntax of the language agents is given by the following grammar. We assume that c and c_i are finite constraints (i.e., elements) of the underlying constraint system.

$$A, B ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B \mid A||B \mid \exists x\, A \mid \mathsf{p}(x)$$

Intuitively, the skip agent does nothing; $\mathsf{tell}(c)$ adds the constraint c to the shared store. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ asks the store for some information. It also models the non-deterministic behavior. More specifically, it checks whether the store satisfies the constraints c_i and non-deterministically chooses which branch is to be executed, provided its condition c_i is satisfied. In case no condition c_i is entailed, then the choice agent *suspends* (it will be executed again in the following time instant). The conditional agent now c then A else B executes A if the store satisfies c, otherwise executes B. Note that, due to the partial nature of the stored information, the fact that c is not satisfied does not imply that $\neg c$ is satisfied. $A \parallel B$ executes the two agents A and B in parallel following the *maximal parallelism* model. The $\exists x\, A$ agent is used to define the variable x local to the process A. Finally, $\mathsf{p}(x)$ is the procedure call agent where x denotes the set of parameters of the process p.

The notion of time is interpreted in the language following the idea that updates and consults to the store takes one time unit. Therefore, only the tell, choice and procedure call agents consume time.

Similarly to cc, the store in the original model of tccp behaves monotonically. Thus, it is not possible to change the value of a given variable along the time. This problem can be solved by using *streams*. For instance, we write $X = [1|Z]$ to denote that a variable X records the current value 1 and Z represents the future values of X.

To ease the tasks related to streams manipulation, we use the modified computation model for tccp presented in [1], where the notion of global store was replaced by the notion of *structured store*. The structured store consists of a sequence of stores where each store of the sequence contains only the constraints added in the corresponding time instant ith.

We can see in Figure 1 the operational semantics of the language, borrowed from [1], that is slightly different from the original model [10] due to the new computational model based on structured stores. The semantics is given by means of a transition relation between configurations $\langle A, st\rangle$, composed by an agent A and the associated store st (at a specific time instant).

Let us describe some of the semantic rules. The first rule **R1** describes how the tell agent at time instant t, augments the store st by adding the constraint c and then skipping. Therefore, the constraint c will be available to other agents from the time instant $t + 1$. Rule **R2** states that A_j is executed in the following time

R1 $\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} c\rangle_{t+1}$

R2 $\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$ if $0 \leqslant j \leqslant n, st \vdash_t c_j$

R3 $$\frac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$$ if $st \vdash_t c$

R4 $$\frac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$$ if $st \nvdash_t c$

R5 $$\frac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$$ if $st \vdash_t c$

R6 $$\frac{\langle B, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$$ if $st \nvdash_t c$

R7 $$\frac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B', st' \sqcup st''\rangle_{t+1}}$$

R8 $$\frac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \not\longrightarrow}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B, st'\rangle_{t+1}}$$

R9 $$\frac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x\, A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x\, A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$$

R10 $\langle \mathsf{p}(x), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$ if $\mathsf{p}(x) : -A \in D$

Fig. 1. Operational semantics of the tccp language.

unit whenever st entails the condition c_j. Regarding the conditional agent, **R3** models that, in case that the agent A with the current store st is able to evolve into the agent A' and a new store st' and st satisfies c, then A' is executed in the following time instant with the computed store st'. **R7** models the evolution of the parallel agent: if A with store st is able to evolve into A' with a new computed store st', and also B with store st is able to evolve into an agent B' with st'', then $A'||B'$ is run in the following time instant with the store resulting from the conjunction of st' and st''. The rest of rules are interpreted in a similar way.

3 An authentication protocol example

To illustrate our approach, we consider the simplified version of the Needham-Schroeder protocol public key authentication protocol [21]. This authentication protocol allows principals A (*Alice*) and B (*Bob*) to communicate via the interchange of secret nonces which besides to serve as nonces, serve as authenticators. It is based on the use of public keys cryptography and nonces, and aims to guarantee confidentiality and authentication, i.e., the nonce received by A from B and the nonce received by B from A must be knew only by them. Let us show the informal protocol definition:

1. $A \rightarrow B : \{A, N_A\}_{K_B}$
2. $B \rightarrow A : \{N_A, N_B\}_{K_A}$
3. $A \rightarrow B : \{N_B\}_{K_B}$

The protocol begins when A sends to B a message encrypted with B's public key K_B which contains the identifier of A, and the nonce N_A generated by A. B decrypts that message by using his private key and sends to A a message, encrypted with the public key of A, which contains the received nonce N_A and a new generated nonce N_B. A decrypts the message by using her secret key. Since the message contains the nonce N_A, A can deduce that the message has recently been sent by B. Finally, A sends the confirmation message to B, so that B can infer that he is communicating to A (he had previously sent N_B).

This protocol was defined relying upon the assumption of perfect cryptography and that principals do not divulge secrets. However, it allowed the following man-in-the-middle attack discovered by Lowe in [19] where I_X denotes I impersonating X:

1. $A \rightarrow I : \{A, N_A\}_{K_I}$
2. $I_A \rightarrow B : \{A, N_A\}_{K_B}$
3. $B \rightarrow A : \{N_A, N_B\}_{K_A}$
4. $A \rightarrow I : \{N_B\}_{K_I}$
5. $I_A \rightarrow B : \{N_B\}_{K_B}$

The man-in-the-middle attack describes how an intruder can discover a secret nonce. In the attack, A initiates a protocol run with I, who (impersonating A) starts a second run of the protocol with B. In other words, the intruder asks B to initiate a communication session saying that he is A. More precisely, the attack follows the following steps: First, A sends his name and a nonce to I. Later, I, impersonating A, sends to B the received message. Then, B, thinking that the received message comes from A, sends to her the received nonce and a new one generated by him (corresponding to the second step of the protocol). When A receives the message sent by B, she thinks that it comes from I (the second step of the first protocol run), and sends back to I the confirmation message that contains the nonce generated by B to A. Finally, I sends to B the same confirmation message. At the end of the attack, B thinks he is communicating to A, which is false.

4 Specifying the Needham-Schroeder protocol in tccp

This section presents how we can specify in a general way the role of each participant of a protocol, and the model for the intruder. As a running example, we show how to formally specify the Needham-Schroeder protocol. We can divide the specification process into three parts:

1. The representation of the concepts involved in a protocol by means of certain terms;

2. The encoding of the behavior of each participant of a protocol in a `tccp` declaration;
3. The codification of the *(hostile) environment*, a process that controls the protocol execution by following the Dolev-Yao attacker model [12], as a `tccp` declaration.

The specifications obtained from the first and third phases can be used for the specification of other protocols. In [14], we described the analysis of the Otway-Rees symmetric key authentication protocol [23]. For the second phase, the example that we describe should serve as a guideline to transform the informal definition of a protocol to the corresponding `tccp` program.

4.1 Some specific constraints

Let us describe the concepts appearing in the informal protocol definition by means of terms which in our approach are handled as constraints.

We represent the private knowledge acquired by the members of a protocol with the term `know(`A`,`K`)` where A is the identifier of a protocol participant, and K is a stream that stores the information. For instance, `know(`*alice*`,[a(`b`),` `n(`nN_A`), n(`nN_B`)|`T`])` states that the agent *alice knows* that b is the identifier of a protocol participant (identified by the term `a/1`), and also knows the nonces nN_A and nN_B (`n/1`).

Messages involved in the protocol are represented by a term of the form `msg(`*ItemList*`)`, and when they are posted to the network, a term `chn(`*Msg*`,`*State*`)` is defined. The content of a message *ItemList* is a list of elements. Each element is represented by a term of the form `enc(`*Key*`,`*Content*`)`, stating that the information in *Content* has been encrypted with the key *Key*. The information is represented as a list of atoms following the classical notation for the `cons` constructor of lists. The list of atoms may contain participant's identifiers, nonces, etc... In `chn(`*Msg*`,`*State*`)`, the variables *Msg* and *State* represent the sent message and its status. The status that may be that the message has traveled the network or channel but it has not been processed yet (it remains non-instantiated) or that the message has been already processed by the environment (it is instantiated to the value `ok`). We adopt an asynchronous model for message passing. This means that, when a principal sends a message, it is stored in the channel to be processed later. The term `rcv(`*Msg*`,`*State*`)` is used by the environment to deliver a message to the corresponding recipient. The variable *State* is instantiated to `ok` when the expected principal has processed the delivered message.

Finally, the terms `blk(`*Msg*`)` and `cnt(`*Content*`)` are used by the intruder to state that a message has been blocked, and the information being the content of a message, respectively.

The constraint system. In addition to the representation of the information to be handled during the protocol run, some operations may be implemented in the underlying constraint system. Therefore, we assume that the constraint system provides the following functionality. When invoked, $new(N_A)$ generates a fresh

(random) value for the given variable N_A. This is used for the nonce generation. In our specifications, when we use this function, the variable passed as argument is never instantiated.

The function *free(Var)* returns a boolean value. It is used to check whether, given the current store, the given variable *Var* is instantiated or not. To recover certain information from a given stream, *recover(Stream,Info)* is defined. This function unifies the value matching *Info* in the stream. We can also use the auxiliary function *find(Stream,Info)* that checks whether *Info* can be recovered (returning the value *true*) or not (*false*). To determine this, the function checks if *Info* unifies with, at least, one of the values stored in the given stream. Note that, in contrast, *recover* returns the resulting substitution. For example, a call to *recover(*`[ a(`b`), n(`nN_A`), n(`nN_B`)|`T`] ,` `a(`Ag`))* would return $Ag = b$.

4.2 Encoding principals into tccp

Let us now show how the participants of the protocol can be specified. The specification of principals must be redefined for each different protocol, and it is modeled as a tccp declaration.

The tccp declaration modeling the initiator `A` is shown below.

```
init (A,B) :-
  ∃ N_A,S_A,S_1,S_A1,N_B,S_2,S_3,S_A2
    (tell(N_A = new(N_A)) ||
    (tell(know(A,S_A)) ||
     ask(true)->
        (tell(chn(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1)) ||
        (tell(S_A = [(a(B),n(N_A)) | S_A1]) ||
         ask(rcv(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2) ∧ free(S_2))->
            (tell(rcv(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2) ∧ free(S_2)) ||
            (tell(S_2 = ok) ||
             ask(true)-> (tell(chn(msg(enc(k(B),cons(N_B,nil))),S_3)) ||
                          tell(S_A = [n(N_B) | S_A2])))))))).
```

Following the steps in the informal protocol description, the participant generates a new nonce, stored in N_A, and states S_A as the stream that contains her private knowledge. Then, by means of a tell agent, she sends to the channel the message of the first protocol step. This means that she sends to B her identifier and the generated nonce, both things encrypted with the public key of B. This is modeled by `chn(msg(enc(k(`B`),cons(`A`,cons(`N_A`,`nil`)))),`S_1`)`. In parallel, she updates her private knowledge S_A with the identifier of the responder and the generated nonce. Again in parallel, a choice agent will be executed for detecting when A receives the second message of the protocol (characterized by the constraint `rcv(msg(enc(k(`A`),cons(`N_A`,cons(`N_B`,`nil`)))),`S_2`)` $\wedge$ `free(`S_2`)`, and `free(`S_2`)` ensures that the message has not been processed). When the condition

holds, by means of the tell agents, she recovers in N_B the nonce generated by B and sets S_2 to ok (to ensure that the message is processed just one time). Then, she sends to B the confirmation message that contains the recovered nonce, and updates her private knowledge with such nonce.

The tccp declaration modeling the responder B is shown below.

```
resp (B) :-
  ∃ A,N_A,S_1,N_B,S_B,S_2,S_B1,S_3,S_B2
    ask(rcv(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1) ∧ free(S_1))->
       (tell(rcv(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1) ∧ free(S_1)) ||
       (tell(S_1 = ok) ||
       (tell(N_B = new(N_B)) ||
       (tell(know(B,S_B)) ||
        ask(true)->
           (tell(chn(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2)) ||
           (tell(S_B = [(a(A),n(N_A),n(N_B)) | S_B1]) ||
            ask(rcv(msg(enc(k(B),cons(N_B,nil))),S_3) ∧ free(S_3))->
               (tell(rcv(msg(enc(k(B),cons(N_B,nil))),S_3) ∧ free(S_3)) ||
               (tell(S_3 = ok) ||
                ask(true)-> tell(S_B = [secret(N_B) | S_B2])))))))))).
```

The execution of the participant B is suspended until the constraint in the first choice agent rcv(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1) ∧ free(S_1) is satisfied. This situation models the moment in which he has received the first message of the protocol. Then, by means of the tell agents, he recovers in A and N_A the identifier of the initiator A and the nonce generated by her, respectively; sets S_1 to the value ok, generates the new nonce N_B and states that S_B is being to contain his private knowledge. After this, the specification sends to the channel the second message of the protocol. In particular, B sends to the principal whose identifier has been recovered from the received message the received nonce and a new one generated by him, all this encrypted with the public key of the target principal. Finally, he updates his private knowledge with the recovered information and the generated nonce. After having sent the second message, B waits until the constraint rcv(msg(enc(k(B),cons(N_B,nil))),S_3) ∧ free(S_3) of the choice agent holds, which means that he has received the last message of the protocol. Then, he sets S_3 to the value ok and updates his private knowledge by storing the fact that the nonce previously generated by him N_B is secret.

4.3 Encoding the Environment into tccp

The design of protocols turns out problematic even assuming perfect cryptography. The problem is mainly due to the fact that principals communicate over a network controlled by a malicious agent (an intruder) who can intercept, analyze, and modify messages, being thus able to carry out malevolent actions.

In this section, we describe the specification of the network where the protocols are executed. We do not specify the intruder as a specific principal, but we encode

it within the environment. The resulting specification allows the actions from the Dolev-Yao intruder model [12].

The environment is defined as a cycle where, during each iteration, a message is processed depending on whether the constraint (chn(M,S) $\wedge$ free(S)) of the first choice agent holds, which means that there is a message to process, or not. The tell agents after the condition recover in M the sent message, sets S to the value ok (to avoid processing a message twice) and states that S_I is going to store the intruder private knowledge. Then, one of the labeled actions ($1a$ to $1d$) is non-deterministically chosen to be executed. Finally, the environment is executed recursively in parallel.

```
environment(I) :- ∃M,S,SI,Si (
  ask(chn(M,S) ∧ free(S)) ->
     (tell(chn(M,S) ∧ free(S)) ||
     (tell(S = ok) ||
     (tell(know(I,SI)) ||
  1a ((ask(true) -> tell(SI = [blk(M) | Si]) +
  1b  (ask(true)-> ∃S1 tell(rcv(M,S1)) +
  1c  (ask(true) ->
          now(M = msg(enc(k(I),_))) then
             ∃C,X,S1 (tell(M = msg(enc(k(I),C))) ||
                       ask(true) -> (tell(SI = [cnt(C) | Si]) ||
                                     (tell(recover(SI,a(X))) ||
                                      ask(true)->
                                         tell(rcv(msg(enc(k(X),C)),S1)))))
          else skip +
  1d   ask(true) ->
          now(find(SI,blk(_))) then
             ∃M',S1 (tell(recover(SI,blk(M'))) ||
                     ask(true) -> tell(rcv(M',S1)))
          else skip))) ||
      ask(true) -> environment(I))))))).
```

Let us describe the labeled actions $1a$ to $1d$. We have labeled the code to improve readability of this description. The code in $1a$ models the situation in which the environment blocks the communication and as a consequence the intruder updates his private knowledge with the given term blk. $1b$ models the correct transmission of the message. The message is labeled with the term rcv being S_1 a fresh variable. Actions in $1c$ specify the case when the given message is encrypted with the public key of the intruder. Then, by using some tell agents, he recovers in C the content of such message and the identifier of the principal who is trying to communicate with him. Then, he composes a message with the recovered information and sends it to the honest principal whose identifier was previously recovered. Finally, $1d$ models the replication of message. In case that the intruder can recover from his private knowledge a message which has been blocked before, then he may deliver it.

To summarize, the *blocking of communication* capability of the Dolev-Yao intruder [12] is modeled by the code excerpt labeled with $1a$; The *message intercepting* capability is modeled by the fact that the environment is able to select

the message to be processed. 1c implements the *message composing/decomposing* capability, and finally, the ability of *message replaying* is modeled by actions 1b and 1d.

5 Experimental results

In this section, we illustrate how it is possible to analyze the protocol by using the interpreter for the language and some capabilities from the specification language Maude. tccpInterpreter has been implemented in Maude. It models the tccp formalism including the underlying constraint system, agents, and its operational semantics. The constraint system includes the functionality required for the protocol execution. Then, given the specification of a tccp program we can simulate the behavior of such program following the semantics of the language and, taking advantage of the Maude capabilities, we can also carry out certain kind of reachability analysis. The interested reader can consult [7,8] for a more detailed documentation about Maude.

5.1 The interpreter

Let us first introduce some concepts used to represent the main tccp entities in Maude. The main function `runs` takes as argument the program to be executed, and natural number that states a threshold for the maximum number of time units that we want for the execution of the program (in case we want to make the execution finite). A configuration (state) of the system is encoded by using the constructor symbol `<` $P1$`,`$P2$`,`$P3$`,`$P4$ `>` $\{P5\}$, where $P1$ represents the given program, $P2$ represents the structured store, $P3$ contains the list of variables present in the store, $P4$ is a boolean constraint used to control the execution of certain agents, and $P5$ is the given threshold.

We denote the store as a set of constraints separated by the symbol `,,`. The constructor symbol `empty` represents the empty store. Then, a structured store is represented by a store together with a natural number (between braces). The natural number represents the current time instant (for example, `(('X > 5,,'X < 10)` $\{2\}$`)` is the store at time instant 2). We use the arrow `=>` to separate each store in the sequence representing a structured store. Terms are written as `[`$Term$`]`, for instance,`['a(`A`)]` is the term for a participant identifier.

The underlying constraint system. To refine the verification process, we use the auxiliary function `getGlobalStoreFromStrStoreList` which, given a structured store, returns a unique store with all the information stored so far. This transformation facilitates the action of consulting the store. Moreover, we implement some auxiliary functions in the underlying constraint system. These functions are `getStreamOfTrmKnowInStore`, `getExpFromStream` and `consultContent`. The first one returns the stream that contains, in a given store, the private knowledge of a given principal (whose identifier is given in the term `know`). The second, returns the value (N_B) that is currently labeled as `secret` in a given stream

(in this case, K_B models the private knowledge of 'b). Finally, the function consultContent checks if the values of a given stream, recovered from the given store, contain the given expression (N_B).

5.2 Simulation

Let us show an excerpt of the trace that describes the correct behavior of the Needham-Schroeder protocol. Consider the tccp program modeling the specification of the Needham-Schroeder protocol presented in Chapter 4 and whose initial agent is (init('a,'b) || (resp('b) || environment('i))). For readability, we use the variable DS instead of the whole code specifying the set of declarations of the considered program presented in Chapter 4. It can be seen that, the initial agent initiates a communication of the participant 'a with 'b. To verify the given program in the interpreter, we must invoke the following instruction.

```
search runs ({DS . (init('a,'b) || (resp('b) || environment('i)))} , 22)
  =>* < TpPg , StL , VrLt , Bl > {Nt} such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    K_A := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
    K_B := getStreamOfTrmKnowInStore (TpSt , ['know ('b,'_)]) ∧
    K_I := getStreamOfTrmKnowInStore (TpSt , ['know ('i,'_)]) ∧
    N_B := getExpFromStream (K_B , ['secret ('_)] , TpSt) ∧
    consultContent (K_A , N_B , TpSt) == true' ∧
    consultContent (K_I , N_B , TpSt) == false' .
```

By using the *search* command, we can explore the reachable state space in different ways. The term in the left hand side of the symbol =>* is the initial state in the execution of the tccp program. The right hand side of the symbol specifies the final state of the execution. The symbol =>* searches for a proof consisting of none, one, or more steps that reaches the final state. The final state which is composed by some variables following the structure for configurations presented above. TpPg represents the given program, StL represents the resulting structured store, VrLt represents the corresponding list of variables, Bl is a boolean constraint and Nt represents the given threshold. With the command such that, we filter the expected result. In this case, we establish that the resulting structured store StL must contain 1) the term secret/1 in the private knowledge of the responder 'b, which means that the protocol ended; 2) the value, labeled as secret by 'b, in the private knowledge of the initiator 'a; and finally 3) the value, labeled as secret by 'b must be known just by 'a.

The following LTL property summarize the previous instruction. It states that there exist a state of the system in which the nonce N_B generated by B is only known by A:

$$\Diamond \exists\, K_A, K_B, K_I, N_B \, (\texttt{know('a,}K_A\texttt{)} \wedge \texttt{know('b,}K_B\texttt{)} \wedge \texttt{know('i,}K_I\texttt{)} \wedge \textit{find}(K_B, \texttt{['secret(}N_B\texttt{)]}) \wedge \textit{find}(K_A, N_B) \wedge \neg \textit{find}(K_I, N_B))$$

As one can see below, the specified *search* command computes one solution at state 60. We know that the protocol has been completed since the expected nonce has been declared `secret`. Thus, `'a` is able to complete the protocol with the responder `'b` at time instant 21. Since we are dealing with structured stores, we can observe which information has been added at each time instant.

```
Solution 1 (state 60)
states: 61 rewrites: 324834 in 620ms cpu (627ms real) (523925 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'b),,('B' :=' 'b),,('I :=' 'i)){1}) =>
(([know('A,'SAi)],,('NAi :=' 'nNAi)){2}) =>
(('chn(['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])],'Si1)],,
  ('SAi :=' [['a('B)],['n('NAi)] | 'SAi1])){3}) => (empty{4}) =>
(([know('I,'SI)],,('Si1 :=' 'S),,('S :=' 'ok),,
('M :=' ['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])])){5}) =>
(('rcv('M,'S1)],,('I' :=' 'I)){6}) => (empty{7}) =>
(([know('B','SBr)],,('Ar :=' 'A),,('NAr :=' 'NAi),,('NBr :=' 'nNBr),,
('S1 :=' 'Sr1),,('Sr1 :=' 'ok)){8}) =>
(('chn(['msg(['enc(['k('Ar)],['cons('NAr,['cons('NBr)])])])],'Sr2)],,
  ('SBr :=' [['a('Ar)],['n('NAr)],['n('NBr)] | 'SBr1])){9}) =>
(empty{10}) =>
((('M' :=' ['msg(['enc(['k('Ar)],['cons('NAr,['cons('NBr)])])])]),,
  ('S' :=' 'ok),,('SI :=' 'SI'),,('Sr2 :=' 'S')){11}) =>
(('rcv('M','S1')],,('I'' :=' 'I)){12}) => (empty{13}) =>
(((NBi :=' 'NBr),,('S1' :=' 'Si2),,('Si2 :=' 'ok)){14}) =>
(('chn(['msg(['enc(['k('B)],['cons('NBi)])])],'Si3)],,
  ('SAi1 :=' [['n('NBi)] | 'SAi2])){15}) => (empty{16}) =>
((('M'' :=' ['msg(['enc(['k('B)],['cons('NBi)])])]),,('S'' :=' 'ok),,
  ('SI :=' 'SI''),,('Si3 :=' 'S'')){17}) =>
(('rcv('M'','S1'')],,('I''' :=' 'I)){18}) => (empty{19}) =>
((('S1'' :=' 'Sr3),,('Sr3 :=' 'ok)){20}) =>
('SBr1 :=' [['secret('NBr)] | 'SBr2]){21}
```

We have shown how to interpret the output for an execution. Let us now demonstrate how to look for an attack in the protocol. Similarly to the first case, we have to specify an initial and final state. The idea is to introduce a *bad* final state, i.e., a state where an attack has occurred. In this case, the final state says that both the honest principal `'a` and the intruder `'i` know the secret nonce generated by `'b`.

```
search runs ({DS . (init('a,'i) || (resp('i) || (environment('i) ||
                    ask(true)-> (init ('i,'b) || resp('b)))))} , 26)
  =>* < StL > such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    K_A := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
    K_B := getStreamOfTrmKnowInStore (TpSt , ['know ('b,'_)]) ∧
    K_I := getStreamOfTrmKnowInStore (TpSt , ['know ('i,'_)]) ∧
    N_B := getExpFromStream (K_B , ['secret ('_)] , TpSt) ∧
    consultContent (K_A , N_B , TpSt) == true' ∧
    consultContent (K_I , N_B , TpSt) == true' .
```

The following LTL property summarize the previous instruction. It states that there exist a state of the system in which the nonce N_B generated by B is known by A and I:

$$\Diamond \exists K_A, K_B, K_I, N_B (\texttt{know('a,}K_A\texttt{)} \wedge \texttt{know('b,}K_B\texttt{)} \wedge \texttt{know('i,}K_I\texttt{)} \wedge \textit{find}(K_B, \texttt{['secret(}N_B\texttt{)]}) \wedge \textit{find}(K_A, N_B) \wedge \textit{find}(K_I, N_B))$$

We can see that at time instant 25, the responder 'b stores that the nonce generated by him is secret, thus the protocol run has finished. We can also see that during the execution, it has been stored the knowledge gained by the different participants of the protocol. The initiator 'a has also been completed the protocol (to her knowledge). Moreover, note that 'a thinks, from the time instant 3, that the messages she will receive come from 'i , which is true but from the time instant 11, 'b thinks that he is communicating with 'a, which is false.

```
Solution 1 (state 592)
states: 593 rewrites: 259361436 in 591320ms cpu (591321ms real)
                    (438614 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'i),,('B' :=' 'i),,('I :=' 'i)){1}) =>
((['know('A,'SAi)],,('A' :=' 'i),,('B'' :=' 'b),,('B''' :=' 'b),,
  ('NAi :=' 'nNAi)){2}) =>
((['chn(['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])],'Si1)],,
  ['know('A','SAi')],,('NAi' :=' 'nNAi'),,
  ('SAi :=' [['a('B)],['n('NAi)] | 'SAi1]){3}) =>
((['chn(['msg(['enc(['k('B'')],['cons('A',['cons('NAi')])])])],'Si1')],,
  ('SAi' :=' [['a('B'')],['n('NAi')] | 'SAi1']){4}) =>
((('M :=' ['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])]),,
  ('S :=' 'ok),,('SI :=' 'SAi'),,('Si1 :=' 'S)){5}) =>
((('C :=' ['cons('A,['cons('NAi)])])],,('I' :=' 'I)){6}) =>
((('SAi1' :=' [['cnt('C)] | 'Si]),,('X :=' 'B'')){7}) =>
((['rcv(['msg(['enc(['k('X)],'C)])],'S1)],,('S' :=' 'ok),,
  ('M' :=' ['msg(['enc(['k('B'')],['cons('A',['cons('NAi')])])])]),,
  ('SI' :=' 'SAi'),,('Si1' :=' 'S')){8}) =>
((('I'' :=' 'I),,('Si :=' [['blk('M')] | 'Si']){9}) =>
((['know('B''','SBr')],,('Ar' :=' 'A),,('NAr' :=' 'NAi),,
  ('NBr' :=' 'nNBr'),,('S1 :=' 'Sr1'),,('Sr1' :=' 'ok)){10}) =>
((['chn(['msg(['enc(['k('Ar')],['cons('NAr',['cons('NBr')])])])],'Sr2')],,
  ('SBr' :=' [['a('Ar')],['n('NAr')],['n('NBr')] | 'SBr1']){11}) =>
  (empty{12}) =>
((('M'' :=' ['msg(['enc(['k('Ar')],['cons('NAr',['cons('NBr')])])])]),,
  ('S'' :=' 'ok),,('SI'' :=' 'SAi'),,('Sr2' :=' 'S'')){13}) =>
((['rcv('M'','S1')],,('I''' :=' 'I)){14}) => (empty{15}) =>
((('NBi :=' 'NBr'),,('S1' :=' 'Si2),,('Si2 :=' 'ok)){16}) =>
((['chn(['msg(['enc(['k('B)],['cons('NBi)])])],'Si3)],,
  ('SAi1 :=' [['n('NBi)] | 'SAi2]){17}) => (empty{18}) =>
((('M''' :=' ['msg(['enc(['k('B)],['cons('NBi)])])]),,('S''' :=' 'ok),,
```

```
  ('SI''' :=' 'SAi'),,('Si3 :=' 'S'''))){19}) =>
(((('C' :=' ['cons('NBi)]),,('I'''' :=' 'I)){20}) =>
(((('Si' :=' [['cnt('C')] | 'Si''']),,('X' :=' 'B''))){21}) =>
(((['rcv(['msg(['enc(['k('X')],'C')])],'S1'')]){22}) => (empty{23}) =>
(((('S1'' :=' 'Sr3'),,('Sr3' :=' 'ok)){24}) =>
('SBr1' :=' [['secret('NBr')] | 'SBr2']){25}
```

Apart from the solution to the query, Maude shows up the time spent during the computation of each solution and the number of rewrites performed. In the first case, the result is given in `627ms` after rewriting 324834 steps. For the second case, the system spent almost `10 minutes` after 259361436 rewrites. Note that, due to the generality of the model, that does not restricts to the detection of a single attack but is general for all the Dolev-Yao attacks, we have to explore more possible execution paths for the protocol. We think that this performance is also caused due to the non-determinism of some of the parts of the interpreter.

6 Conclusions and Future Work

We have shown how tccp can be used as a suitable language to verify security protocols. We have taken advantage of the underlying constraint system by assuming we have some functions defined at the constraint system level. This allows us to improve the compactness and clarity of the model. Many of the defined components in our model, in particular the environment and the specified constraints, can be reused for the analysis of other protocols. Finally, by using the tccpInterpreter system, which is an interpreter implemented in Maude, we have shown how we can check safety properties on tccp programs to detect vulnerabilities that may lead to attacks.

The tccpInterpreter is publicly available at the following addresses: `http://www.dsic.upv.es/~villanue/tccpInterpreter/` and `http://www.dsic.upv.es/~alescaylle/tccp.html`.

We plan to extend the specification of the `environment`, increasing actions an attacker can perform, for example to include typing attacks. We also plan to improve the performance of the interpreter and to study how to adapt the model-checking technique existing for tccp to this rewriting-based framework.

References

1. Alpuente, M., Gallardo, M.M., Pimentel, E., Villanueva, A.: Verifying Real-Time Properties of tccp Programs. j-jucs **12**(11) (2006) 1551–1573
2. Alpuente, M., Gallardo, M., Pimentel, E., Villanueva, A.: A semantic framework for the abstract model checking of tccp programs. Theoretical Computer Science **346**(1) (2005) 58–95
3. Basin, D., Denker, G.: Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis. In Futatsugi, K., ed.: Electronic Notes in Theoretical Computer Science. Volume 36., Amsterdam, Elsevier Science Publishers (2001)

4. Bella, G., Bistarelli, S.: Soft constraint programming to analysing security protocols (2004)
5. Boer, F.S.d., Gabbrielli, M., Meo, M.C.: A Timed Concurrent Constraint Language. Information and Computation **161**(1) (2000) 45–83
6. Clark, J.A., Jacob, J.L.: A survey of authentication protocol literature. Technical report, Defence Evaluation Research Agency (1997)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Volume 4350 of Lecture Notes in Computer Science. Springer (2007)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Maude Web Site (2009) `http://maude.csl.sri.com/`.
9. Corin, R., Etalle, S., Hartel, P.H., Mader, A.: Timed model checking of security protocols. In: FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering, New York, NY, USA, ACM (2004) 23–32
10. de Boer, F.S., Gabbrielli, M., Meo, M.C.: A Temporal Logic for Reasoning about Timed Concurrent Constraint Programs. In: TIME '01: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01), Washington, DC, USA, IEEE Computer Society (2001) 227
11. Denker, G., Meseguer, J., Talcott, C.: Protocol Specification and Analysis in Maude. In Heintze, N., Wing, J., eds.: Proceedings of Workshop on Formal Methods and Security Protocols, Indianapolis, Indiana (Jun 1998)
12. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols. IEEE Transactions on Information Theory **29**(2) (March 1983) 198–208
13. Falaschi, M., Villanueva, A.: Automatic Verification of Timed Concurrent Constraint Programs. Theory and Practice of Logic Programming **6**(3) (May 2006) 265–300
14. Lescaylle, A.: Master's thesis: The Timed Concurrent Constraint language in practice (December 2009) Available at `http://www.dsic.upv.es/~alescaylle/files/tesis-09.pdf`.
15. Lescaylle, A., Villanueva, A.: Using tccp for the Specification and Verification of Communication Protocols. In: Proceedings of the 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07). (2007)
16. Lescaylle, A., Villanueva, A.: Verification and Simulation of protocols in the declarative paradigm. Technical report, DSIC - Univeridad Politécnica de Valencia (2008) Available at `http://www.dsic.upv.es/~alescaylle/files/dea-08.pdf`.
17. Lescaylle, A., Villanueva, A.: A Tool for Generating a Symbolic Representation of tccp Executions. Electron. Notes Theor. Comput. Sci. **246** (2009) 131–145
18. Lescaylle, A., Villanueva, A.: The tccp Interpreter. Electron. Notes Theor. Comput. Sci. **258**(1) (2009) 63–77
19. Lowe, G.: Breaking and Fixing the Needham-Schroeder public-key protocol using FDR. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer Verlag (1996) 147–166
20. Meadows, C.: Formal Verification of Cryptographic Protocols: A Survey. In: ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology, LNCS, Springer-Verlag (1994)
21. Needham, R., Schroeder, M.: Using Encryption for Authentification in Large Networks of Computers. Communications of the ACM **21**(12) (December 1978) 993–999
22. Olarte, C., Valencia, F.D.: Universal concurrent constraint programing: symbolic semantics and applications to security. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, New York, NY, USA, ACM (2008) 145–150

23. Otway, D., Rees, O.: Efficient and timely mutual authentication. ACM Operating System Review **21**(1) (1987) 8–10
24. Saraswat, V.A.: Concurrent Constraint Programming Languages. The MIT Press, Cambridge, MA (1993)

Implementation and Evaluation of a Declarative Debugger for Java

Herbert Kuchen[1] and Christian Hermanns[2]

[1] University of Münster, Münster, Germany,
kuchen@uni-muenster.de
[2] University of Münster, Münster, Germany,
hermannc@wi.uni-muenster.de

Abstract. We present a declarative debugger for Java. Declarative debugging is a technique widely used in the area of logic and functional logic programming. We demonstrate how this method can be adapted to the object-oriented programming paradigm and show how to tackle crucial implementation problems of this method. We also extend a technique to reduce the number of questions which we developed in former research in this area and show how this technique can be integrated into our declarative debugger and present the key concepts and techniques of its implementation. A further important contribution of this paper is to provide empirical evidence for the usefulness of the technique for reducing the number of questions the user has to answer.

1 Introduction

Object-oriented languages such as Java heavily rely on the concept encapsulation. Encapsulation hides the internal behavior of software components and reduces their mutual dependency. This facilitates their reusability and reduces complexity, making encapsulation a powerful and successful concept for the software development process [1]. Today it is commonly used in the design and implementation phase of software development.

Nevertheless, the concept of encapsulation has rarely been used in the debugging phase of software development. Common debuggers for object-oriented languages as Java still trace the debuggee program in a step by step manner and do not take advantage of component relationships.

Declarative debugging was developed by E. Y. Shapiro [2] for the Logic Programming paradigm. It was later applied to other declarative programming paradigms such as functional [3] and functional logic [4] programming.

In this paper we present a debugging tool which implements a declarative debugging method for the object-oriented language Java. The theoretical foundations of the debugging method were developed in former work [5]. During the debugging session our debugger will identify an erroneous method by asking questions about the correctness of method calls occurring during an erroneous computation, deriving the intended method semantics from the user's answers. Thus, the user can concentrate on "what" a method does (the semantics) and no

longer needs to know "how" a method works internally. The main contribution of this paper is to show how the debugging method can be transformed into a working tool. Furthermore, we present some test results which reveal the behavior of our debugger in practice.

The rest of the paper is organized as follows: In the next section we present a program which we will use as a running example throughout the paper. Section 3 shows how we apply the declarative debugging technique to the Java programming language. Furthermore, it presents a method which reduces the number of questions a user must answer to find a bug. In section 4 we demonstrate how our debugger can be used to find the wrong method in the sample program. Section 5 presents the architecture of our debugger and points out important implementation techniques. In section 6 we evaluate the results of a set of test cases we processed with our debugger. The experiments show the usefulness of our techniques for reducing the number of questions. In section 7 we discuss how our paper is related to other works in this field. Our work ends with section 8, where we conclude and point out future work.

2 Sample Program

We have chosen the well-known *binary tree* data structure [6] which we will use as a running example throughout the paper. The code of the binary tree comprises the classes `Node`, `Tree`, and `TestTree`:

```
1 class Node<K extends Comparable<K>, D> {
2   K fKey;
3   D fData;
4   Node<K, D> fLeft;
5   Node<K, D> fRight;
6
7   Node(K key, D data, Node<K, D> left, Node<K, D> right) {
8     fKey = key;
9     fData = data;
10    fLeft = left;
11    fRight = right;}
12
13  public void insertNode(K key, D data) {
14    if (fKey.compareTo(key) < 0)
15      if (fRight == null)
16        fRight = new Node<K, D>(key, data, null, null);
17      else
18        fRight.insertNode(key, data);
19    else if (fLeft == null)
20      fLeft = new Node<K, D>(key, data, null, null);
21    else
22      fLeft.insertNode(key, data);}
23
24  public D findNode(K key) throws Exception {
25    if (fKey.compareTo(key) < 0)
26      if (fRight == null)
27        throw new Exception(key + " not found");
28      else
29        return fRight.findNode(key);
30    else if (key.compareTo(fKey) > 0)
31      if (fLeft == null)
32        throw new Exception(key + " not found");
33      else
34        return fLeft.findNode(key);
```

```
35    else
36      return fData;}}
```

```
1 public class Tree<K extends Comparable<K>, D> {
2   protected Node<K, D> root = null;
3
4   public void insert(K key, D data) {
5     if (root == null)
6       root = new Node<K, D>(key, data, null, null);
7     else
8       root.insertNode(key, data);}
9
10  public D find(K key) throws Exception {
11    if (root == null)
12      throw new Exception(key+" not found");
13    else
14      return root.findNode(key);}}
```

```
1 public class TestBinaryTree {
2   public static void main(String[] args) throws Exception {
3     Tree<Integer, String> tree = new Tree<Integer, String>();
4     tree.insert(4, "four");
5     tree.insert(3, "three");
6     tree.insert(2, "two");
7     tree.insert(1, "one");
8     System.out.println(tree.find(1));}}
```

This binary tree supports insert and search operations. The intended semantics of the methods is straightforward. The `insert(K key, D data)` and `find(K key)` methods of the `Tree` class delegate each operation to the root node of the data structure. The only special case which needs to be considered here is the empty root. The `insertNode(K key, D data)` method of the `Node` class delegates the insert operation to the right child node if the value of the argument `key` is greater than the value of the field `fKey` of the node whose method is called and to the left child node otherwise. If the selected child node does not exist a new child node is created for this node. The `findNode(K key)` method works alike. The find operation is continued in the right subtree if the value of the argument `key` is greater than the value of the node's field `fKey` and in the left subtree if `key` is smaller. The search is successful if `key` == `fKey` and it fails if the selected subtree is empty. The method `findNode(K key)` contains an error in line 30 which should be "**`else if (key.compareTo(fKey) < 0)`**".

3 The Declarative Debugging Method

3.1 The Computation Tree

To enable the declarative debugging of Java programs we use a computation tree structure which we defined in [5]. Each node of the computation tree will contain information about a particular method call of the computation. Let N be a node of the computation tree containing information about a call of method m. Child nodes of N correspond to method calls occurring in m that have been executed during the computation of the result of N. For example, the computation tree representing the execution of our sample program in Section 2 is shown in the first column of Table 1.

In our computation tree structure a *buggy node* is an invalid node which has no invalid children. The call represented by the buggy node produced a wrong result while it's inputs, i.e. the arguments provided by the parent call and the return values of the child calls, are all valid. Therefore, the debugging technique is correct and guarantees to find the erroneous method.

Method calls in an object-oriented language can produce side effects which are part of the result of a method call. These side effects must be taken into account during validation of a method call. Hence, the data stored at each node of our computation tree will be:

- a fully qualified name identifying the method being called.
- all local variables of the method call, i.e. the arguments of the call and all locally defined variables. In case of a call to a non-static method the "this" reference to the object whose method is being called is considered as the first argument. For each local variable an entry and an exit value will be stored. The entry value is the variables value at the beginning of the method call and can be regarded as part of the method's input. The exit value is the value at the end of the call and belongs to the method's result.
- objects contain a set of fields. For any object referenced by a local variable we need to know the entry and exit values of its fields w.r.t. the considered method call. An array is considered as a special type of object with all its fields of the same type. Note that the fields of referenced objects can in turn reference further objects, spanning a graph of referenced objects. We call this graph the state space which is accessible by the respective method call. The accessible state spaces spanned by the entry and exit values are part of the method's input and the method's result, respectively.
- additionally, the entry and exit values of static class fields should be available because they can be part of the methods input and output as well.
- finally, we need the return value.

The usability of the debugger will depend on a compact and clear representation of the relevant (changed) information.

Note that our approach just identifies an erroneous method. It does not expose an erroneous statement in this method. This would require a combination of declarative and trace debugging, which is out of scope of this paper and will be tackled in future work. If - according to good programming style - a method only consists of a few lines, identifying an erroneous method is already sufficient to find a bug quickly.

3.2 Reducing the Number of Questions

The number of questions asked by our debugger can still be large when it comes to the debugging of complex programs. To reduce the debugging effort we have employed a heuristic to automatically derive answers for *equivalent* questions which we developed in [5].

The definition of equivalent questions, i.e. equivalent method calls, is inspired by the approaches of glass-box testing [7]. During glass-box testing a system of

test cases, i.e. pairs of inputs and corresponding outputs of a component being tested, which in some sense covers the possible control and/or data flows of a tested component is generated. Generally, it is not possible to test all possible control and/or data flows, since there are to many and often infinitely many of them. Thus, a reasonable coverage criterion is regarded as sufficient. We have employed two common coverage criteria: coverage of nodes and edges of the control flow graph (see Figure 1) and coverage of def-use chains [8].

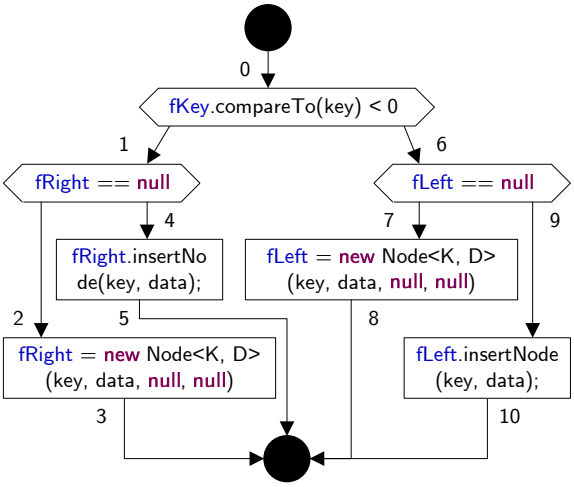

Fig. 1. Control-flow graph of the method `insertNode` in the binary search example. The edge numbers correspond to those in Table 1.

A def-use chain is a triple of a variable, a statement where the variable's value is computed, i.e. defined, and a statement where the variable's value is used. Moreover, the value of the variable must not change between the definition and the use. For example, in class `Node` of our running example the parameter passing in line 13 together with the if statement in line 14 constitute a def-use chain for the variable `key`, denoted by (key,13,14) for short. Other examples for def-use chains are (key,13,18) and (data,13,16). Note that a definition needs not to be textually above a use, in particular in the presence of loops.

Following the ideas of coverage based testing we will define classes of equivalent questions. Two questions are members of the same class of equivalent questions iff their related method calls produced the same coverage. Our debugger will assume that the answers to all questions of the same equivalence class are identical. For example, if a question concerning the soundness of a method call `m(`a_1,...,a_n`)` shall be asked by the debugger and if there was a previous question corresponding to some method call `m(`b_1,...,b_n`)`, where `m(`a_1,...,a_n`)` and `m(` b_1,...,b_n`)` are equivalent w.r.t. the corresponding coverage of the control or data

flow, the second question will not be asked but the result of the first question will be reused.

This heuristic allows our debugger to detect most but not all errors [5]. Thus, we will allow the user to switch off the coverage-based inference of answers if the buggy node could not be found during the debugging process. After the coverage-based inference is disabled, the user will also be asked the questions which were automatically inferred before. This will allow the user to detect most of the errors quickly and easily in the first place. The additional effort to answer equivalent questions is only required if an error could not be detected using answer inference.

Table 1. Computation tree produced by the binary tree example of section 2. The first column contains the hierarchy of method calls, i.e. the computation tree. The second and the third column show the def-use chain and edge coverage produced by the corresponding method calls, respectively. The following abbreviations are used for the variable names of def-use chains: t=tree, rt=root, k=key, d=data, l=left, r=right, fK=fKey, fD=fData, and fL=fLeft.

Method call	Edge coverage	Def-use chain coverage
TestBinaryTree.main()→void	[0,1,3,5, 7,9]	[(t,3,4),(t,3,5),(t,3,6),(t,3,7), (t,3,8)]
⊢Tree.<init>()→void	[0,1]	[]
⊢Tree.insert(4,"four")→void	[0,1,3]	[(rt,4,5), (k,4,6),(d,4,6)]
| └Node.<init>(4,"four",null,null)→void	[0,1]	[(k,7,8),(d,7,9),(f,7,10),(r,7,11)]
⊢Tree.insert(3,"three")→void	[0,2,4]	[(rt,5,6),(k,4,6),(d,4,6)]
| └Node.insertNode(3,"three")→void	[0,6,7]	[(fK,14,15),(k,14,15),(fL,14,20), (k,14,21),(d,14,21)]
| └Node.<init>(3,"three",null,null)→void	*[0,1]	*[(k,7,8),(d,7,9),(f,7,10),(r,7,11)]
⊢Tree.insert(2,"two")→void	[0,2,4]	[(rt,5,6),(k,4,6),(d,4,6)]
| └Node.insertNode(2,"two")→void	[0,8,9]	[(fK,14,15),(k,14,15),(fL,14,20), (fL,14,23),(k,14,23),(d,14,23)]
| └Node.insertNode(2,"two")→void	*[0,6,7]	*[(fK,14,15),(k,14,15), (fL,14,20),(k,14,21),(d,14,21)]
| └Node.<init>(2,"two",null,null)→void	*[0,1]	*[(k,7,8),(d,7,9),(f,7,10),(r,7,11)]
⊢Tree.insert(1,"one")→void	*[0,2,4]	*[(rt,5,6),(k,4,6),(d,4,6)]
| └Node.insertNode(1,"one")→void	*[0,8,9]	*[(fK,14,15),(k,14,15),(fL,14,20), (fL,14,23),(k,14,23),(d,14,23)]
| └Node.insertNode(1,"one")→void	*[0,8,9]	*[(fK,14,15),(k,14,15),(fL,14,20), (fL,14,23),(k,14,23),(d,14,23)]
| └Node.insertNode(1,"one")→void	*[0,6,7]	*[(fK,14,15),(k,14,15),(fL,14,20), (k,14,21),(d,14,21)]
| └Node.<init>(1,"one",null,null)→void	*[0,1]	*[(k,7,8),(d,7,9),(f,7,10),(r,7,11)]
└Tree.find(1)→"four"	[0,2,4]	[(rt,11,12),(k,11,15),(rt,11,15)]
└Node.findNode(1)→"four"	[0,13,14]	[(fK,26,27),(k,26,27),(k,26,32), (fK,26,32),(fD,26,38)]

Table 1 shows the hierarchy of method calls in our running example. Each call is associated with the edges of the control flow graph and the def-use chains it covered. Due to space restrictions the coverage information of a method call does not contain the coverage information of its subcalls. The complete coverage of a method is the union of its own coverage set and the coverage sets of any of its children. If a coverage information is preceded by a * the classification of the method call can be inferred from previous user answers using the respective

coverage criterion. In this case the automatically inferred answers would be the same regardless of the selected coverage criterion. Note that especially for more complex method implementations the sets of inferred answers do in fact differ depending on the coverage criterion we select. In our example any coverage criterion would eliminate the question about method call `Tree.insert(1, "one ")` during the debugging session.

4 Debugging the Sample Program

The debugging session starts when the user runs the `main` method of the `TestBinaryTreeClass` and notices the unexpected output "four" rather than "one". Now the user starts the debugger which repeats the computation producing the computation tree which is partly displayed in the Computation Tree View of Figure 2. The root of the computation tree corresponds to the initial call of the `main` method which is marked by the user as invalid. From this starting point the debugger will select further methods following a *top down* search strategy. Anyhow, the user does not need to follow this strategy and is free to select any method call of the computation tree by himself. In our example we will validate the method calls in the following order proposed by the debugger:

- The call of the constructor `Tree.<init>()`. As this constructor is empty, it is easily detected as valid.
- The next node is the call of `Tree.insert(4, "four")` which is selected in the Computation Tree View of Figure 2. In the Node Explorer View of the same figure we see that the local variable **`this`** has changes in its subtree because it is marked by a yellow (light grey) background color. Its field `root` which is marked red (dark grey) has changed from **`null`** (Old value) to a reference to a `Node` object with id=263 (New value) during the selected method call. The fields `fKey` and `fData` of this object have changed from **`null`** to the values which we want to insert into our data structure. A closer look at the Computation Tree View in Figure 2 tells us that the new `Node` object has obviously been constructed by the only child node, a call to `Node.<init>(4, "four", null, null)`. We mark the `insert` call as valid because the result reflects our expected behavior.
- In the next three steps the debugger asks us to classify the calls `Tree.insert(3, "three")`, `Tree.insert(2, "two")`, `Tree.insert(1, "one")`. The result of all three calls is a new `Node` object which is added to the tree data structure. We can select these nodes in the Computation Tree View and inspect the changes in the subtree of the **`this`** variable in the Node Inspector View as we have done for the first call to the `insert` Method. The observed result does correspond to the intended semantics of this method: each of the new nodes is added as leftmost child to the tree data structure. With automatic answer inference enabled our debugger would skip the question about `Tree.insert(1, "one")`. As indicated by the coverage information of Table 1, this node will be classified as valid based on the classification of `Tree.insert(2, "two")` which produced the same coverage.

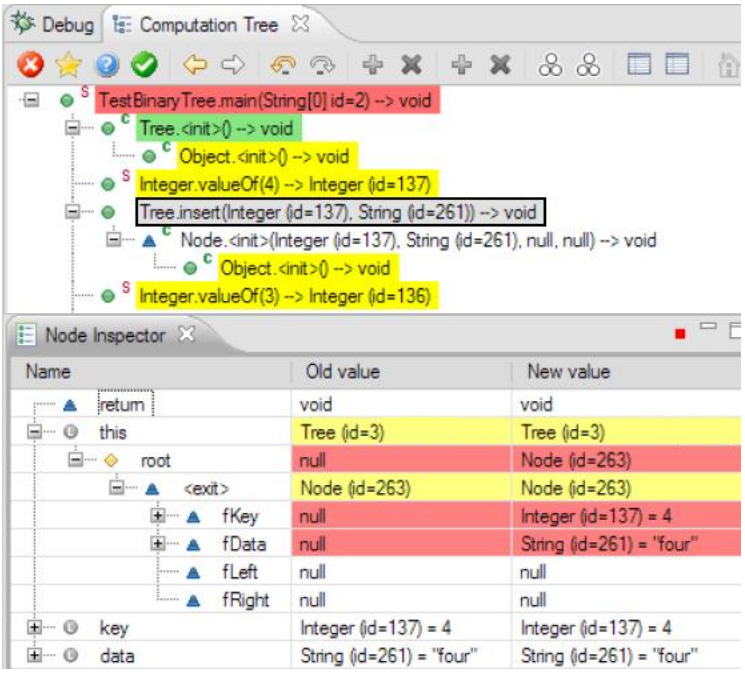

Fig. 2. Screenshot of the declarative debugger user interface. The "Computation Tree" view shows the computation tree representing the execution of the `main` method of the `TestBinaryTree` class of the running example. The "Node Inspector" View shows the values of the local variables of the `insert` method which is selected in the "Computation Tree" view. The columns "Old value" and "New value" show the variable values *before* and *after* execution of the selected method call. Variables whose value has changed are marked by a different background color: red (dark grey) indicates value changes of the variable itself, yellow (light grey) indicates changes in the subtree of a variable.

- Now we are asked to validate `Tree.find(1)`. The binary tree data structure which is stored in the `root` field of the `Tree` object being called does not contain a `Node` with `fKey`=1 and `fData`= "four" which would justify the return value of the method. Hence, this node is invalid. The debugger will continue to look for the buggy node in the subtree of the current node.
- The call of `Node.findNode(1)` will also be defined as being invalid. The `Node` object's left subtree contains the searched (key,value) pair (1,"one"). Hence, the return value "four" of the method call is clearly false.

At this moment the debugger marks the method `findNode(K key)` as buggy, ending the debugging session. Now, the user has to check the method and correct the error.

As we have seen, some of the questions that occur during a debugging session can be complex. Notice however that the same questions will occur implicitly during a debugging session using a normal trace debugger. Moreover, the use of the declarative debugger facilitates the debugging process, allowing the user to compare the input and output values of variables modified during a method call. The directed navigation also helps by reducing the questions to nodes with an invalid parent node, while a trace debugger also traces valid computations. Some additional features of our tool help to further reduce the debugging effort:

- Classes can be marked as trusted before the debugging process starts, just like the Java API packages in our example. This reduces the size of the tree and therefore the number of questions. Furthermore, the user can mark the method associated to any node as *trusted* during the debugging session. Subsequently, all nodes associated to this method are automatically valid.
- State changes of local variables and object fields are marked by different background colors in the "Node Explorer" view as shown in Figure 2. This greatly simplifies the detection of side effects and reduces the effort to classify method calls.
- The computation tree gives the user better overview and understanding of the general control flow of the debuggee program. A feature which helps to reduce debugging effort, but is totally missing in trace debuggers.

5 Implementation

5.1 Architecture

Basically, there are two possible techniques to obtain the required debugging information about the execution of the debuggee program: the Java Platform Debugger Architecture (JPDA) [9] and program transformation. JPDA is an API specified by the Java Virtual Machine Specification which has an event-based architecture. It can be used to obtain events about an executed program such as method-entry, method-exit, and field change events. Because JPDA is part of any Java Virtual Machine implementation it appears to be a convenient and efficient way to implement our debugger. Unfortunately, it is not possible to collect any def-use chain or edge coverage information about an executed program with JPDA. For example, JPDA does not allow to trace read and write operations of local variables, a feature which is necessary to record def-use chain coverage. Hence, we employed program transformation using the ASM framework [10] to obtain all required information about the execution of the debuggee program.

Figure 3 shows the architecture of our declarative debugger. It consists of four basic components: Instrumentation Engine, Recording Engine, Debugging Engine, and the debuggee program. The components are distributed over two separate JVMs, the Debugger JVM and the Debuggee JVM. The Debugging Engine resides in the Debugger JVM, while the Instrumentation Engine and the debuggee program are located in the Debuggee JVM. The Recording Engine is distributed over both JVMs.

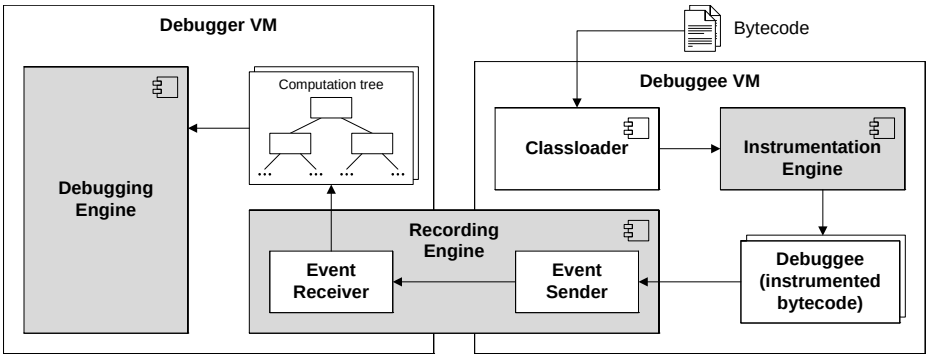

Fig. 3. Architecture of the declarative debugger.

During the class loading process of the Debuggee JVM the bytecode of the loaded class is manipulated by the Instrumentation Engine. After the instrumentation process, the bytecode of the loaded class contains additional instructions which invoke methods of the `EventSender` class.

The methods of the `EventSender` create debugging events which are transmitted to the `EventReceiver`. As the instrumentation process is conducted during the dynamic class loading process at bytecode level, it is transparent from a programmers point of view. In a debugging session a programmer will work with the original source code. This makes the program transformation invisible to him.

During the execution of the debuggee program the sequence of debugging events is collected by the `EventReceiver` which constructs the computation tree from these events.

The current implementation of the `EventReceiver` creates an object model representation of these computation trees in the memory of the Debugger JVM. The virtue of this approach is that it guarantees fast access to the elements of each computation tree for the Debugging Engine. On the other hand, storing complete computation trees in memory is very resource intensive, especially for large computations the memory requirements can be huge. To reduce the memory consumption of our debugger we could store the computation tree in a file or database first and load the required parts into memory on demand. However, this solution has not been implemented yet.

5.2 Data Model

Figure 4 shows the data model of our declarative debugger. The model is represented as a class diagram and contains all the relevant information needed to apply the debugging technique described in section 3. The data model is constructed by the `EventReceiver` class based on the received debugging events during execution of the debuggee program.

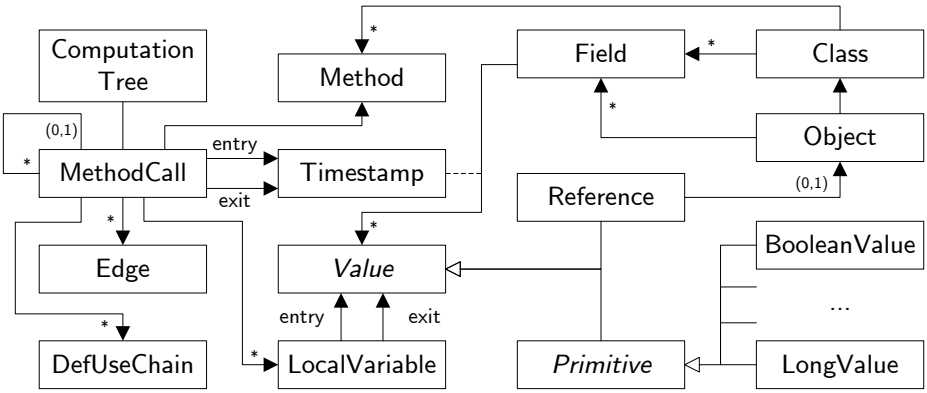

Fig. 4. Class diagram of the data model of the declarative debugger.

The data model contains a `ComputationTree` representing the the execution of the debuggee program. A `ComputationTree` has a root `Method-Call` which is usually the call of the `main` method of the debuggee program. Each `MethodCall` is associated to a specific `Method` and has a set of local variables and a return value. Note that the arguments of the method call as well as a possible "this" reference to the object whose method is called are also treated as local variables. A set of covered def-use chains and a set of covered edges of the control flow graph is also stored with each method call. These sets are used to identify equivalent method calls (section 3.2). Moreover, each method call has a parent call except for the root call which has no parent call and 0 or more child calls. These relations build a computation tree as described in section 3.1.

Additionally, a method call has two associated timestamps. The first timestamp indicates the time at the beginning of the method call and the second timestamp indicates the time at the end of the method call. The timestamps are needed to determine the values of class and object fields at the beginning and the end of a method call. A timestamp is an ordinal value which must meet the following conditions: Let E_a be the entry timestamp and X_a be the exit timestamp, then $E_a < X_a$ must hold for any method call a. Let b be a method call occurring *after* the execution of a, i.e. after the computation of a is finished, then $X_a < E_b$ must hold. Furthermore, let c be a child call of a, i.e. a call which occurred *during* the execution of a then $E_a < E_c < X_c < X_a$ must hold. This way the timestamps are guaranteed to be consistent with the method call hierarchy. Note that the timestamps for a method call can be computed from the method call's position in the computation tree in $O(n)$, where n is the number of calls in the computation tree. To avoid recomputation of these values, they are stored with the method calls.

Each local variable has an entry and an exit value representing the variable's value at the beginning and the end of a method call. A value can either be a primitive value or a reference value. Primitive values represent the well-known

primitive values of the JVM (boolean, int, etc.). The reference value can either reference an object or be a null reference. An object belongs to a class and contains a set of fields. A field can have an arbitrary number of values each associated with a timestamp of the computation. When we need to compute the values of a field at method entry and method exit, we can use the method call's timestamps to determine the field's values. If the values are stored chronologically, a field's value at some timestamp can be determined in $O(\log v)$ using binary search, where v is the number of values stored values for that field.

Note that the computation of the relevant entry and exit values for a method call is a recursive process. The process starts by computing the entry and exit values for all local variables. After that, the process continues with the computation of the entry and exit values of the fields of the objects referenced by the previously determined values. The entry and exit values of these fields can in turn reference further objects. This process is recursively applied to all referenced objects which have not been visited yet. A method call can also read and write static class fields. For this reason, the process is also applied to the fields associated to relevant classes of the computation.

Please note that the described data model is a logic view to the data recorded during the execution of a debuggee program. Although the current implementation stores the recorded events in an object model which is based on the described class class diagram, there are other ways to store the collected data. The model can also be stored in files or a database. As indicated before this would help to reduce the memory usage of our debugger.

6 Test Results

To show the practicability of our debugging technique and our debugger implementation, we have applied our debugger to a set of selected test programs. For each of the programs which originally contained no errors we created several test samples. Each sample was generated from the original program by changing one distinct line of the source code such that the resulting test sample would contain a common programming error. This way we have created a set of 38 test samples out of 6 different programs. Using our declarative debugger we have determined the number of questions a user would have to answer, i.e. how many method calls he has to evaluate, before the erroneous method is found by the debugger. The debugging process has been conducted three times for each error sample. Once with automatic answer inference disabled and twice with automatic answer inference enabled using def-use chain coverage and edge coverage, respectively. Counting the number of questions we have to answer for each of the three debugger settings allows us to evaluate their usefulness.

Table 2 shows the test results. Column one indicates the name of the program which was subject to debugging. For each program we have manually constructed several test cases as described before. The number of test cases we created this way is shown in column two. The values in column 3 indicate the average size of the generated computation tree. Column 4 shows the average number of trusted

method calls in the computation tree which are automatically marked as valid. The next 3 columns contain the average number of answers necessary to find the erroneous method. Each column corresponds to a coverage criterion we employed to automatically infer answers. The coverage criteria are: no coverage criterion, def-use chain coverage, and edge coverage. Depending on the test program either def-use chain coverage or edge coverage is the inference strategy which eliminates the most questions. Overall edge coverage performs best, asking 10,53 questions on average, while the other strategies yield 17,29 (no inference) and 11,66 (def-use chain). The ratio of the visited search space given in columns 8-10 is calculated by dividing the average number of answers by the average number of untrusted method calls in the computation. On average we have to validate 27% of the methods of the computation tree if we use no inference strategy, 18% if we use def-use chain coverage, and 17% if we use edge coverage.

Table 2. Results of test cases processed with the declarative debugger.

	# of	Avg. # of		Avg. number of			Ratio of visited			Relative	
	Test	method calls		answers			search space			savings	
Program	Cases	Total	Trusted	None	DUC	Edge	None	DUC	Edge	DUC	Edge
Avl	10	142,0	74,5	13,90	12,40	12,60	0,21	0,18	0,19	0,11	0,09
Binary Tree	5	138,4	94,4	11,80	11,60	11,60	0,27	0,26	0,26	0,02	0,02
B Tree	8	214,1	137,3	16,13	13,88	14,25	0,21	0,18	0,19	0,14	0,12
Heap sort	5	35,0	1,0	12,20	10,40	9,20	0,36	0,31	0,27	0,15	0,25
Hash table	5	162,0	66,6	32,40	10,00	5,00	0,34	0,10	0,05	0,69	0,85
Total Avg.	33	138,3	74,8	17,29	11,66	10,53	0,27	0,18	0,17	0,33	0,39

The last two columns indicate the relative number of questions that can be eliminated with answer inference in comparison to debugging without answer inference. Depending on the test program and the coverage the percentage of eliminated questions varies between a minimum of 2% and a maximum of 85%, with an average of 33% for def-use chain and 39% for edge coverage.

7 Related Work

The idea to apply declarative debugging outside the declarative programming paradigm is not new. In 1998 Shahmehri and Fritzson presented an approach for declarative debugging of the imperative language *Pascal* [11] which was further developed by the same authors in [12]. The main difference of our approach w.r.t. these earlier proposals is that Java is a language much more complex than Pascal. The declarative debugging of programs including objects and object states introduces new difficulties.

There are several approaches which use an execution history to locate bugs in the debuggee program. In a first step these methods trace and record the complete execution of the debuggee program. In a second step the recorded

information is used to identify errors in the debuggee program. For example, JavaDD [13] follows a query-based approach, storing events occurring during the debuggee program execution in a deductive database. The database can be queried to retrieve the states of different program entities (variables, threads, etc.) at different moments of the computation history to infer where the bug is located. Another approach is omniscient debugging [14] which can trace the execution of the debuggee program back and forth in time. In contrast to this approaches our debugger concentrates on the logic of method calls, storing them in a structured way, i.e. the computation tree. This allows our debugger to guide the debugging process, deducing the wrong method from user answers.

Hoon-Joon Kouh et al. [15] propose a debugging technique for Java which combines algorithmic and step-wise debugging. In contrast to our work they neither present a tool implementation nor do they present a solution to display side effects in an accessible form. Furthermore, they use slicing techniques to reduce the size of the computation tree, while our tool uses coverage information.

Our paper is based on previous work [5] we have done in this field. In this paper we extend our previous ideas and show their practicability by presenting a declarative debugger implementation. Our debugger enhances the functionality of a former prototype and includes an implementation of the automated answer inference presented in section 3.2. Furthermore, we test and evaluate our debugger using various test scenarios.

8 Conclusions and Future Work

We have presented a tool which enables the declarative debugging of Java programs. A major difference to the situation found in declarative languages is that Java enables side effects. Besides the parameters and the result of a method call these side effects must be presented to the user for validation of the method calls. This requires a well-designed user interface which does not overstrain the tester with too much information, but allows to expand the required pieces of the state space on demand. This includes the demand driven navigation of the object graph which is accessible from local variables and object fields.

The major advantage of declarative debugging compared to conventional debugging is that it works on a higher level of abstraction. The tester is relieved from the task to inspect the state space after each instruction starting from some break point. By answering questions about the soundness of some method calls the user can concentrate on the semantics. Furthermore, compared to a trace debugger the overall debugging effort is not greater. During trace debugging the user will answer questions about the correctness of method calls implicitly while observing changes of the state space in a step by step manner.

A particular novelty of our approach is the usage of code-coverage criteria such as def-use chain coverage and coverage of the edges of the control-flow graph which we use to define the equivalence of method calls. This notion of equivalence enables us to reduce the amount of questions significantly.

We have conducted a number of tests where we have used our debugger to find the erroneous method. The results show that our debugger is suitable for practical application. Note that real-world applications are debugged component by component such that the search space for the component under consideration is manageable. Moreover, these tests confirm that our notion of equivalent method calls clearly reduces the number of questions.

In the future we plan to integrate trace debugging functionality into our declarative debugger. If a user is uncertain about the correctness of a method call tracing its execution back and forth in time will provide further information which helps to classify method calls. Hence, this hybrid debugging approach will have the benefit that it combines the advantages of both debugging methods.

References

1. Booch, G.: Object-Oriented Analysis and Design with Applications (3rd Edition). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
2. Shapiro, E.Y.: Algorithmic Program DeBugging. MIT Press, Cambridge, MA, USA (1983)
3. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. J. Funct. Program. **11**(6) (2001) 629–671
4. Caballero, R., Rodríguez-Artalejo, M.: A declarative debugging system for lazy functional logic programs. Electronic Notes in Theoretical Computer Science **64** (2002)
5. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of java programs. Electron. Notes Theor. Comput. Sci. **177** (2007) 75–89
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. The MIT Press (2001)
7. Pressman, R.S.: Software Engineering: A Practitioner's Approach. fifth edn. McGrap-Hill (2001)
8. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (August 2006)
9. Sun Microsystems: Java platform debugger architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/` (2009)
10. Object Web: Asm. `http://asm.ow2.org/` (2009)
11. Shahmehri, N., Fritzson, P.: Algorithmic debugging for imperative languages with side-effects. In Third International Workshop CC '90 Schwerin, FRG, O..., ed.: Compiler Compilers. Volume 477., Berlin, Germany, Springer (1991) 226–227
12. Fritzson, P., Shahmehri, N., Kamkar, M., Gyimothy, T.: Generalized algorithmic debugging and testing. ACM Lett. Program. Lang. Syst. **1**(4) (1992) 303–322
13. Girgis, H.Z., Jayaraman, B.: Javadd: a declarative debugger for java. Technical report, Department of Computer Science and Engineering, University at Buffalo (7 2006)
14. Lewis, B.: Debugging backwards in time. CoRR **cs.SE/0310016** (2003)
15. Kouh, H.J., Kim, K.T., Jo, S.M., Yoo, W.H.: Debugging of java programs using hdt with program slicing. In: ICCSA (4). (2004) 524–533

Author Index