FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03)

Jörg Striegnitz, Kei Davis[*] (Eds.)

FZJ-ZAM-IB-2003-11

August 2003

(last change: 22.08.2003)

([*]) Modelling, Algorithms, and Informatics Group, CCS-3 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Table of Contents

SOUL and Smalltalk - Just Married

Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis

Kris Gybels*

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Elsene, Belgium
kris.gybels@vub.ac.be

1 Introduction

The *Smalltalk Open Unification Language* is a Prolog-like language embedded in the object-oriented language Smalltalk [5]. Over the years, it has been used as a research platform for applying logic programming to a variety of problems in object-oriented software engineering, some examples are: representing domain knowledge explicitly [3]; reasoning about object-oriented design [15, 14]; checking and enforcing programming patterns [11]; ; checking architectural conformance [16] and making the crosscuts in Aspect-Oriented Programming more robust [6]. These examples fit in the wider research of *Declarative Meta Programming*, where SOUL is used as a meta language to reason about Smalltalk *code*.

Recently, we explored a different usage of SOUL in connecting business rules and core application functionality [2], which involves reasoning about Smalltalk *objects*. We found we had to improve on SOUL's existing mechanism for interacting with those objects because it was not transparent: it was clear from the SOUL code when rules were invoked and when messages were sent to objects, vice-versa solving queries from methods was rather clumsy. Ideally we would like to achieve a *linguistic symbiosis* between the two languages: the possibility for programs to call programs written in another language as if they were written in the same [8, 13]. Such a transparent interaction would make it easy to selectively change the paradigm parts of an application are written in: if we find that a Smalltalk method is better written as a logic rule we should be able to replace it as such without having to change all messages invoking that method.

We will here take a historical approach to describing the SOUL/Smalltalk symbiosis. We would like to provide an insight into our motivation for and approach to achieve the symbiosis by contrasting three distinct stages in its evolution. In a first stage, SOUL was developed as a direct Prolog-derivate with some additional mechanisms for manipulating Smalltalk objects as Prolog values. In a second and third stage we explored alternative mechanisms and a more Smalltalk-fitting syntax for SOUL. Interestingly, when we performed a survey of other combinations of object-oriented and logic programming we found we could

* Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

easily categorize their approaches into one of our three "stages". The following sections discuss the stages in detail and the "Related Work" section at the end briefly discusses the survey.

2 Stage 1: Escaping from SOUL

The interaction mechanism found in the original SOUL can best be characterized as an escape mechanism. But before we go into this, let us make some general points about this version of SOUL:

Syntax: We assume readers are familiar with Prolog, the differences with this language and SOUL in this stage are:

- **Variable notation:** in Prolog, variables are written as names starting with a capital letter, in SOUL they are written as names preceded with a question mark, thus `Something` translates to `?something`.
- **List notation:** in Prolog, square brackets (`[ ]`) are used to write lists, these are replaced with angular brackets in SOUL (`< >`).
- **Rule notation:** the 'if' operator `:-` linking conclusion to conditions is replaced with the `if` operator in SOUL.

The combination of Smalltalk's Meta-Object Protocol and SOUL's embedding in Smalltalk lead to the insight that the simplest way to let SOUL programs reason about Smalltalk code is to give them access to the meta-objects directly. For this reason there are additional differences with Prolog:

Values: any Smalltalk object (not just the meta-objects) can be bound as a value to a logic variable.

Syntax: the Smalltalk term, a snippet of Smalltalk code enclosed in square brackets `[ ]`. The Smalltalk code can contain logic variables wherever Smalltalk variables are allowed.

Semantics: when Smalltalk terms are encountered as conditions in rules, they are "proven" by executing the Smalltalk code. The return value should be a boolean, which is interpreted as success or failure of the "proof". Smalltalk terms can also be used as arguments to conditions, then they are evaluated and the resulting value is used as the value of the argument. Unification deals with Smalltalk objects as follows: two references to an object unify only if they refer to the same object.

Primitive predicates: a primitive predicate `generate` can be used to generate elements of a Smalltalk collection as successive solutions for a variable.

The example set of rules in figure 1 are taken from SOUL's library for Declarative Meta Programming and show how Smalltalk terms are used. A predicate `class` is defined which reifies class meta-objects into SOUL; two different rules are defined for it to deal efficiently with different argument binding patterns. The `subclass` predicate expresses that two classes are related by a direct subclassing

```
class(?x) if
  var(?x),
  generate(?x, [ System allClasses ])
class(?x) if
  nonvar(?x),
  [ ?x isClass ]

subclass(?super, ?sub) if
  class(?sub),
  equals(?super, [ ?sub superclass ])

hierarchy(?root, ?child) if
  subclass(?root, ?child)
hierarchy(?root, ?child) if
  subclass(?root, ?direct),
  hierarchy(?direct, ?child)
```

Fig. 1. Example rules defining predicates for reasoning about Smalltalk programs.

```
argumentArray := Array with: (Array with: #x with: someClass).
evaluator := SOULEvaluator eval: 'if hierarchy(?x, ?y)' withArgs: argumentArray.
results := evaluator allResults.
ysolutions := results bindingsForVariableNamed: #y.
```

Fig. 2. Code illustrating how the SOUL evaluator is called from Smalltalk and how the results are retrieved.

relationship when one is the answer to the `subclass` message sent to the other. The `hierarchy` predicate extends this to indirect subclassing relationships.

The example rules are indicative for the way SOUL interacts with Smalltalk in this stage: the use of Smalltalk terms is limited to a small collection of predicates such as `class` and `subclass`, which are organized in the so-called "basic layer". Other, more high-level predicates such as `hierarchy` make use of the predicates in the basic layer to interact with Smalltalk objects. This organization avoids pollution of the higher-layer predicates with explicit escape code[1]. In a way, the basic layer provides a gateway between the two languages by translating messages to predicates and vice-versa.

The other direction of interaction, from Smalltalk to SOUL, is done through explicit calling of the SOUL evaluator with the query to be evaluated passed as a string. Figure 2 illustrates how the `hierarchy` predicate is to be called. On the second line, an evaluator object is created by sending the message `eval:withArgs:` to the `SOULEvaluator` class, the message is passed the query

[1] Another reason why this is done is to make the higher-layer predicates less dependent on Smalltalk, so that they may later be used when reasoning about code in other OO languages [4].

to evaluate and variable bindings as arguments. The variable bindings are passed as an array of variable-object pairs. In the example, the logic variable `?x` will be bound to the value of the Smalltalk variable `someclass`, so the query will search for all child classes of that class. These child classes will then be bound as solutions to the variable `?y`. These solutions can be retrieved by sending an `allResults` message to the evaluator object, which returns a result object. The result object then needs to be sent the message `bindingsForVariableNamed` to actually retrieve the bindings, which are returned as a collection.

3 Stage 2: Predicates as Messages

A second stage of SOUL-Smalltalk interaction, which we reported on at a previous multi-paradigm programming workshop [1], aimed at providing more of a transparent interaction. Our motivation then was especially to improve on the way Smalltalk programs can invoke queries, and do it in a way that would provide *linguistic symbiosis*. To do so, we tried to map invocation of predicates more directly to the concept of sending a message.

The term linguistic symbiosis refers to the ability for programs to call programs written in another language as if they were written in the same. Having this ability would also imply that transparent replacement is possible: replacing a "procedure" (= procedure/function/method/...) in the one language with a "procedure" in the other, without having to change the other parts of the program that make use of that "procedure". In fact, the term was coined in the work of Ichisugi et al. on an interpreter written in C++ which could have all of its parts replaced with parts written in the language it interprets. Such usage of linguistic symbiosis to provide reflection was further explored in the work of Steyaert [13].

While these earlier works provided us with solutions, we also had an added problem: the earlier works dealt with combining two languages founded on the object-oriented paradigm, while we aimed at combining an object-oriented and a logic language. The earlier works dealt with mapping a message in the one language to a message in the other, while we needed to map messages to queries.

To provide a mapping of messages and queries, we had five issues to resolve:

Unbound variables: how does one specify in such a message that some arguments are to be left unbound? The concept of 'unbound variables' is foreign to Smalltalk.

Predicate name: how is the name of the predicate to invoke derived from the name of the message?

Returning multiple variables: how will the solutions be returned when there are multiple variables in the query?

Returning multiple bindings: if there are multiple solutions for a variable, how will these be returned?

Receiver: which object will the message be sent to?

Message	Query
`Main add: 1 with: 2 to: 3`	`if Main.add:with:to:(1,2,3)`
`Main add: 1 with: 2`	`if Main.add:with:to:(1,2,?res)`
`Main add: 1`	`if Main.add:with:to:(1,?y,?res)`
`Main add`	`if Main.add:with:to:(?x,?y,?res)`
`Main addwith: 2`	`if Main.add:with:to:(?x,2,?res)`
`Main addwithto: 3`	`if Main.add:with:to:(?x,?y,3)`
`Main addwith: 2 to: 3`	`if Main.add:with:to:(?x,2,3)`
`Main add: 1 withto: 3`	`if Main.add:with:to(1,?y,3)`

Table 1. Mapping a predicate to messages

We combined the solution for the first two issues by assuming that predicate names, like Smalltalk messages, would be composed of keywords, one for each argument. To specify which variables to leave unbound we adopted a scheme for combining these keywords into a message name from which that specification can be derived. To invoke a predicate from Smalltalk one would write the message as: the name of the first keyword, optionally followed by a colon if the first argument is to be bound and a Smalltalk expression for the argument's value, then the second keyword, concatenated to the first if that one was not followed by a colon, and again itself followed by a colon if needed for an argument and so on for the other keywords until no more keywords need to follow which take an argument. This is best illustrated with an example. Table 1 shows the 2^3 ways of invoking a predicate called `add:with:to:` and the equivalent query in SOUL.

For the issue of needing a receiver object for the message, we mapped layers to objects stored in global variables. Because in Smalltalk classes are also objects stored in global variables, this has the effect of making a predicate-invoking message seem like a message to a class. The basic layer is for example stored in `Basic`.

We proposed two alternative solutions to the issues of returning bindings. The first was simply to return as result of the message a collection of collections: a collection containing for each variable a collection of all the bindings for that variable. The alternative consisted of returning a collection of message forwarding objects, one for each variable. Sending a message to such a forwarding object would make it send the same message to all the objects bound to the variable. The idea was to provide an implicit mechanism for iterating over all the solutions of a variable, very much how like SOUL can backtrack to loop over all the solutions for a condition. This however lead to matters such as whether forwarding objects should also start backtracking over solutions etc., so it was discarded as a viable solution. We coined the term *paradigm leak* to refer to this problem of concepts "leaking" from one paradigm to the other.

We also used the predicate and message mapping to replace SOUL's earlier use of Smalltalk terms. Instead of using square brackets to escape to Smalltalk for sending a message, the same message can now be written more implicitly as an invocation of a predicate in an object "pretending to be a SOUL module".

Here, the reverse of the above translation happens: SOUL will transform the predicate to a Smalltalk message by associating the arguments of the predicate to the keywords in its name. The predicate's last argument will be unified with the result of the actual message send. Take the following example:

```
if Array.with:with:with:(10,20,30, ?instance), ?instance.at:(2,?value)
```

The first condition in the example query will actually be evaluated by sending the message `with: 10 with: 20 with: 30` to the class `Array`. The result of that message is a new `Array` instance, which will be bound to the variable `?instance`. In the second condition, the message `at: 2` will be sent to the instance and the result, 20 in this case, will be bound to the variable `?value`.

While in this second-stage SOUL mixing methods and rules is entirely transparent from a technical standpoint, it is obvious which code is intended to invoke what to a human interpreter. Technically there is no more need in SOUL for an escape mechanism, and the same language construct is used to invoke rules and messages. Similarly in Smalltalk, queries no longer have to be put into strings to let them escape to SOUL and can just be written as message sends. However, a Smalltalk programmer would frown when seeing messages such as `addwith: 2 to: 3`. Furthermore, he would probably guess that the result of that message would be the value 5, instead it will be a collection with a collection containing the value 5. The keyword-concatenated predicate names in SOUL also lead to awkward looking programs in that language.

4 Stage 3: Linguistic Symbiosis?

The next, and currently last, stage in the SOUL-Smalltalk symbiosis uses a new syntax for SOUL to avoid the clumsy name mappings from the previous stage. For this stage we also had a specific application for the symbiosis in mind, business rules [2], which influenced its development in certain respects. One difference is that previously we wanted to allow Smalltalk programs to call the existing library of SOUL code-reasoning predicates, while for supporting business applications the idea is rather to use SOUL to write new rules implementing so-called business rules of the application. This also implies another shift: reasoning about (business) objects rather than meta objects.

In the new syntax predicates look like message sends. Let us illustrate with an example, figure 3 contrasts the classic `member` predicate with its new `contains:` counterpart.

The second rule for `contains:` can be read declaratively simply in Prolog-style as "for all `?x`, `?y` and `?rest` the `contains:` predicate over `<?y | ?rest>` and `?x` holds if ....". A declarative message-like interpretation could read "for all `?x`, the answer to the message `contains: ?x` of objects matching `<?y | ?rest>` is true if the answer of the object `?rest` to `contains: ?x` is true." Both interpretations are equivalent, though the second one is really the basis for the new symbiosis.

```
member(?x, <?x | ?rest>).
member(?x, <?y | ?rest>) if
  member(?x, ?rest).

<?x | ?rest> contains: ?x.
<?y | ?rest> contains: ?x if
  ?rest contains: ?x.
```

Fig. 3. Comparison of list-containment predicate in classic and new SOUL syntax.

```
?product discountFor: ?customer = 10 if
  ?customer loyaltyRating = ?rating &
  ?rating isHighRating
```

Fig. 4. Example of a rule using the equality operator.

Because messages can return values other than booleans, we added another syntactic element to SOUL to translate this concept to logic programming. The equality sign is used to explicitly state that "the answer to the message on the left hand side of `=` is the value on the right hand side". Figure 4 shows an example.

The new syntax has a two-fold impact on how the switching between Smalltalk and SOUL occurs. It is no longer necessary to employ a complicated scheme with concatenation of keywords to get the name of a predicate. Another is that there is no more mapping of objects to SOUL modules and vice-versa, modules were dropped from SOUL as the concept of having a "receiver" for a predicate now comes as part of the message syntax.

A Smalltalk program no longer has to send a message to a SOUL module "pretending to be an object" to invoke a query. Instead, a switch between the two languages now occurs as an effect of method and rule lookup: we changed Smalltalk so that when a message is sent to an object and that object has no method for it, the message is translated to a query. In SOUL, when a rule is not found for the predicate of a condition, the condition is translated to a message. This new scheme makes it much easier and much more transparent to actually interchange methods and rules.

The translation of queries and messages is straightforward and we'll simply illustrate with another example. Figure 5 shows a price calculation method on a class `Purchase` which loops through all products a customer bought and sums up their total price minus a certain discount. When the `discountFor: customer` message is sent to the products, Smalltalk will find no method for that message, so it will be translated to the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

Where `?arg1` and `?arg2` are already bound to the objects that were passed as arguments to the message. When the query is finished, the object in `?result`

```
Purchase instanceVariables: 'shoppingBasket customer'

Purchase>>totalPrice

  | totalPrice discountFactor |

  totalPrice := 0.
  shoppingBasketContents do: [ :aProduct |
    discountedFactor := (100 - (aProduct discountFor: customer)) / 100
    totalPrice := totalPrice + (discountFactor * aProduct price)
  ]
```

Fig. 5. Example price calculation method on Purchase class

is returned as result of the "message". This returning of results is actually a bit more involved, we'll discuss it further in the next section.

For the inverse interaction, we can take the `loyaltyRating =` condition in the `discountFor:` rule (fig. 4) as an example. For a small business the loyalty rating of a customer can simply be stored as a property of the customer object which can be accessed through the `loyaltyRating` message. In that case, SOUL will find no rule for the "predicate" `loyaltyRating` and will translate the condition simply to the message `loyaltyRating` which is then sent to the customer object in the variable `?customer`. After it returns, the result of the message is unified with the variable `?rating`. Of course, for a bigger business we might want to replace the calculation of `loyaltyRating` with a set of more involved business rules which we'd prefer to implement with logic programming, for example "a high rating is given to a customer when she has already spent a lot in the past few months". With the transparent symbiosis such a replacement is easy to do.

5 Limits and Issues

At the end of the "Stage 2" section, we remarked that our solution then was only technically transparent, it was rather obvious to a programmer which code was intended to invoke which paradigm. In the previous section we demonstrated that this is now much less the case, it is fairly easy now to interchange methods and rules without this becoming obvious. There are however limits to this interchanging and there are still subtle hints that may reveal what paradigm is invoked. These limits and issues stem from differences in programming style between the object-oriented and logic paradigms.

One important style difference between the paradigms is the way multiplicity is dealt with. In logic programming, there is no difference between using a predicate that has only one solution and one that has multiple solutions. In object-oriented programming there is an explicit difference between having a message return a single object or a collection of objects (even, or especially, if

```
?child ancestor = ?parent if
  ?child parent = ?parent.
?person ancestor = ?ancestor if
  ?person parent = ?parent,
  ?parent ancestor = ?ancestor
```

Fig. 6. Rules expressing the ancestor relationship between Persons

```
Person instanceVariables: 'name parent'

Person>>parent
   ^ parent

Person>>name
  ^ name

Person>>printOn: stream

  name , ' descendant of ' printOn: stream.
  self ancestor do: [ :ancestor |
     ancestor name , ' and ' printOn: stream
  ]
```

Fig. 7. Instance variables and some methods of the Person class

there's only one object in that collection). This difference leads to an issue in how results are returned from queries to Smalltalk, and one in how predicates and messages are named.

When a Smalltalk message invokes a SOUL query and the query has only one solution, should the solution object be simply returned or should a singleton collection with that object be returned? The invoking method may expect a collection of objects, which would then just happen to contain just a single item, or it may generally be expecting there to be only one result. It is difficult for SOUL however to know which is the case. To deal with this we made SOUL return single solutions in a `FakeSingleItemCollection` wrapper. The `FakeSingleItemCollection` class implements most of the messages expected of collections in Smalltalk, any other messages are forwarded to the object that is being wrapped. There is thus an "automatic adaptation" to the expectations of the invoking method.

Plurality, or lack thereof, in the names of predicates and messages can cause some programming style difficulties. Figures 6 and 7 illustrate the modeling of persons and their ancestral relations through a class and some logic rules. Invoking these rules from the `printOn:` method is however awkward: it is quite natural for a logic programmer to write the relationship as "ancestor" even though there will be multiple ancestors for each Person, the object-oriented programmer would however prefer to write the plural "ancestors" to indicate that a

collection of results is expected. One solution to this problem is to implement a rule for `ancestors` which simply maps to `ancestor`, this would however defeat the purpose of having an automatic mapping of messages and queries. A potential solution could be to take this style difference into account when doing the mapping by adding or removing the suffix *-s* when needed.

When comparing the stage 2 and stage 3 symbiosis, stage 3 may seem more limited in the variables that can be left unbound when invoking queries from Smalltalk. In stage 2 the mapping of predicate names to message names implicitly also indicated which variables to leave unbound, while in stage 3 the mapping of messages to queries only leaves unbound the result variable, the one on the right hand side of the equality sign in the query. Actually, we did implement a means for leaving other variables unbound as well. We changed the way Smalltalk deals with temporary variables to allow for the following code to be written:

```
| products customers discounts |

discounts := products discountFor: customers
```

Normally the Smalltalk development environment would warn that this code uses temporary variables before they are assigned. Now however, the message `products discountFor: customers` will result in the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

Where all of `?arg1`, `?arg2` and `?result` are left unbound. When the query is finished, the result of the message will be as described earlier and additionally the temporary variables `products` and `customers` will also be assigned the solutions of the variables `?arg1` and `?arg2`.

This leaving unbound of temporary variables is however another example of a paradigm leak, it is quite unnatural code for a Smalltalk programmer to write. We consider it as something that should be used with care and preferably avoided. While in stage 2 the equivalent mechanism seemed most necessary because the motivation was to allow access to all *existing* SOUL predicates, our focus shift to implementing *new* business rules makes it less necessary: its better to design the rules differently. Nevertheless many of the rules will be designed to be used from other rules, not to be replaced with methods and callable in a multi-way fashion. On occasion, these rules may need to be used directly from a method, so we kept the unbound temporaries mechanism in place.

There is also a limitation in leaving arguments unbound the other way around: when translating a condition to a message, all of its arguments are expected to be bound. SOUL will currently generate an error otherwise. It would be possible though to at least deal with the "receiver" argument of the condition in a more logic-like way: when it is unbound, SOUL could send the message to some random object from memory which support a method for the message. If the message's result is true, the object is a solution for the "receiver" variable. On backtracking all of the other objects supporting the message would be tried. A problem here would be the accidental invocation of object-mutating messages

due to polymorphism: when posing the query `?game draw` we may simply be interested in all chess games that ended in a draw but may wind up also drawing all graphical objects on the screen. In practice though this may not be so much of a problem as normally the messages invoked from SOUL would have a keyword "is" or "has" in their name because they are written as invocation of predicates, and it is a convention normally applied by Smalltalk programmers as well.

6 Related work

We examined several existing systems which were designed or could be used for business rule development and in which object-oriented and logic programming are combined [2]. The interaction mechanisms we encountered fit in one of three categories similar to the three stages we discussed here: use of an escape mechanism, some explicit mapping of predicates and methods or a syntactic and semantic integration of the two languages. We limit our discussion here mostly to a few systems that aim for the third category as well.

NéOpus also extends Smalltalk with logic programming, though with production-rule based logic rather than proof-based logic [12]. Rules consist of conditions and actions, rather than conclusions, which are respectively expressed as boolean messages to objects and state-changing messages to objects. The concept of a "conclusion" as something separate from a direct effect on the state of objects is thus dropped. Rules are also not invoked through queries, but rather are triggered by changes in the state of objects and there is no backtracking to generate multiple solutions. This means that some of the issues we had to deal with do not occur in NéOpus: the problems of mapping predicates and methods, returning of multiple results etc. Pachet in fact argues against adding backward chained inferencing to NéOpus because he finds there's a contradiction between the desire to use the OO language to express rules in and allow backward chaining [12], which may come down to our issues. Note that we made the rule language resemble the object-oriented one as closely as possible and needed to allow for symbiosis, we did not simply use the OO language directly to express rules. Besides what form of chaining to use, Pachet also discusses other questions which we had to resolve as well. Most importantly what happens to pattern matching and object encapsulation. Often in logic programming a data structure is accessed directly through unification of its constituent parts. In some of the other systems we examined, like CommonRules [7] , this is still done this way by mapping objects to predicates with an argument for each instance variable. In SOUL, as in NéOpus, we chose to uphold object encapsulation and only allow accessing objects through message sending.

LaLonde and Van Gulik used Smalltalk's reflection to turn ordinary methods into backtracking methods [10]. They built a small framework[2] to support the backtracking methods. Most important in there is a message which makes its

[2] Small enough to have the full code listed in their paper.

calling method return with a certain value but remembers the calling point, the method can then be made to resume execution from that point on. This is achieved by exploiting Smalltalk's ability to access the execution stack from any method. The backtracking takes care of undoing changes to local variables, though not to instance variables and globals . Local variables are thus used to simulate logic variables, but they are assigned rather than unified and there is no simulation like our unbound temporaries for calling methods with some arguments left unbound, so backtracking methods are no full simulation of logic rules. Despite the similarities in the use of Smalltalk expressions, programming in this system seems quite different from programming in symbiotic SOUL.

Kiev [9] extends Java with logic rules, which can be added directly to classes and called through message sending. To call a rule with unbound arguments, one passes an empty wrapper object as argument which will then be bound by the rule. A new for-construct can be used to iterate over all solutions. There is no equivalent for our equality operator construct, calling a rule from a method as a message always returns a boolean to indicate success or failure. This is a subtle but important difference with symbiotic SOUL: returning objects from rules requires the use of sending a message with unbound arguments, making calling rules not as transparent.

7 Summary and Conclusions

We presented the history of a combination of Smalltalk with a logic language. Three distinct stages appeared in its evolution of the interaction between the two languages which we also encountered in studying other combinations of object-oriented and logic programming: a stage where the languages could bind each other's values to variables and manipulate these values by "escaping" to the other language, a stage where the escape mechanism was made more transparent by an automatic mapping of predicates and methods and the current final stage in which the syntax of the logic language has been adapted to that of the host language to allow not only for technical but programming style transparency as well. The aim was to achieve a linguistic symbiosis so that methods and rules can be easily and transparently interchanged. This is not just of theoretical interest but has an application in the development of business rule applications: an existing application without business rule separation may need to be turned into one that does, or new developments in the policies of the business may make it more interesting to turn methods into rules.

We compared our earlier and current solution for such issues as how to map messages and queries, return multiple results from a query to Smalltalk etc. There are unfortunately still some minor issues to resolve such as how to deal properly with the difference in use of plurality in names between the two paradigms and avoiding the invocation of state-changing messages. Nevertheless we have found the current version of symbiotic SOUL to be a great improvement over previous versions.

References

[1] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards a linguistic symbiosis of an object-oriented and logic programming language. In Jörg Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2002.

[2] Maja D'Hondt and Kris Gybels. Linguistic symbiosis for the automatic connection of business rules and object-oriented application functionality. (to appear), 2003.

[3] Maja D'Hondt, Wolfgang De Meuter, and Roel Wuyts. Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering*, 1999.

[4] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. In *Proceedings of the European Smalltalk User Group's conference*, 2003. (Conditionally accepted).

[5] Adele Goldberg and Dave Robson. *Smalltalk-80: the language*. Addison-Wesley, 1983.

[6] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.

[7] IBM. Business rules for electronic commerce: Project at IBM T.J. Watson research, 1999. http://www.research.ibm.com/rules/.

[8] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: a reflective object-oriented concurrent language without a runtime kernel. In *IMSA'92 International Workshop on Reflection and Meta-Level Architectures*, 1992.

[9] Maxim Kizub. *Kiev language specification*, July 1998. http://www.forestro.com/kiev/kiev.html.

[10] Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility for Smalltalk without kernel support. In *Proceedings of the conference on Object-Oriented Languages, Systems and Applications*. ACM Press, 1988.

[11] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, 2001.

[12] Francois Pachet. On the embeddability of production rules in object-oriented systems. *Journal of Object-Oriented Programming*, 8(4), 1995.

[13] Patrick Steyaert. *Open Design of Object-Oriented Languages*. PhD thesis, Vrije Universiteit Brussel, 1994.

[14] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 1998*, 1998.

[15] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[16] Roel Wuyts and Kim Mens. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 1999*, 1999.

Syntax sugar for FC++: lambda, infix, monads, and more

Brian McNamara and Yannis Smaragdakis

College of Computing
Georgia Institute of Technology
http://www.cc.gatech.edu/~yannis/fc++/
lorgon,yannis@cc.gatech.edu

Abstract. We discuss the FC++ library, a library for functional programming in C++. We give an overview of the library's features, but focus on recent additions to the library. These additions include the design of our "lambda" sublanguage, which we compare to other lambda libraries for C++. Our lambda sublanguage contains special syntax for programming with monads, which we also discuss in detail. Other recent additions which we discuss are "infix function syntax" and "full functoids".

1 Introduction

FC++[7, 8] is a library for functional programming in C++. We have recently added a number of new features to the FC++ library, most notably an expression template library for creating a *lambda* sublanguage. The lambda sublanguage contains special syntax for programming with *monads* in the style of Haskell. We focus our discussion on the design of this portion of the library (Section 5 and Section 6), but begin with a run-down of the features of FC++ (Section 2 and Section 3) as well as some important implementation details (Section 4).

2 Overview

In FC++, programmers define and use *functoids*. Functoids are the FC++ representation of functions; we will discuss them in more detail in Section 4. The latest version (v1.5) of the FC++ library supports a number of useful features, including

- higher order, polymorphic functoids ("direct" functoids)
- lazy lists
- a large library of functoids, combinators, and monads (most of which duplicate a good portion of the Haskell Standard Prelude[2])
- currying
- infix functoid syntax
- dynamically-bound functoids ("indirect" functoids)

– a small library of effect combinators
– interfaces to C++ Standard Library data structures and algorithms via iterators
– ways to transform methods of classes and normal C++ functions into functoids
– reference-counted "smart" pointers for memory management (used internally by, e.g., our lazy list data structure)

We'll briefly discuss each of these features in the next section. Later on we will discuss

– special syntax to mimic functional language constructs, including *lambda*, *let*, and *letrec*, as well as *do*-notation and *comprehensions* for arbitrary monads

in detail.

The FC++ library is about 9000 lines of C++ code, and is written with strict conformance to the C++ standard[4], which makes it portable to all of the major brands of compilers.

3 Short Examples of various features

FC++ functoids can be simultaneously higher order (able to take functoids as arguments and return them as results) and polymorphic (template functions which work on a variety of data types). For example, consider the library function `compose()`, which takes two functoids and returns the composition:

```
// compose( f, g )(args)  ==  f( g(args) )
```

We could define a polymorphic functoid `addSelf()`, which adds an argument to itself:

```
// addSelf( x )  ==  x + x
```

We could then compose `addSelf` with itself, and the result would still be a polymorphic functoid:

```
int x = 3;
std::string s = "foo";
compose( addSelf, addSelf )( x )      // yields 12
compose( addSelf, addSelf )( s )      // yields "foofoofoofoo"
```

Section 4 describes the infrastructure of these "direct functoids", which enables this feat to be implemented.

FC++ defines a lazy list data structure called `List`. `List`s are lazy in that they need not compute their elements until they are demanded. For example, the functoid `enumFrom()` takes an integer and returns the infinite list of integers starting with that number:

```
enumFrom( 1 )     // yields infinite list [1, 2, 3, ...]
```

A number of functoids manipulate such lists; for instance `map()` applies a functoid to each element of a list:

```
map( addSelf, enumFrom( 1 ) )  // yields infinite list [2, 4, 6, ...]
```

The FC++ library defines a wealth of useful functoids and data types. There are named functoids for most C++ operators, like

```
plus(3,4)       // 3+4      also minus, multiplies, etc.
```

There are many functoids which work on `Lists`, including `map`. Most of the `List` functions are identical those defined in Haskell[2]. Additionally, a number of basic functions (like the identity function, `id`), combinators (like `flip`: `flip(f)(x,y)==f(y,x)`), and data types (like `List` and `Maybe`; `Maybe` will be discussed in Section 6) are designed to mimic exactly their Haskell counterparts. We also implement functoids for such C++ constructs as constructor calls and `new` calls:

```
construct3<T>()(x,y,z)   // yields T(x,y,z)
new2<T>()(x,y)           // yields new T(x,y)
```

and many more (some of which are described below).

Functoids are curryable. That is, we can call a functoid with some subset of its arguments, returning a new functoid which expects the rest of the arguments. Currying of leading arguments can be done implicitly, as in

```
minus(3)      // yields a new function "f(x)=3-x"
```

Any argument can be curried explicitly using the placeholder variable `_` (defined by FC++):

```
minus(3,_)    // yields a new function "f(x)=3-x"
minus(_,3)    // yields a new function "f(x)=x-3"
```

We can even curry all N of a function's arguments with a call to `curryN()`, returning a *thunk* (a zero-argument functoid):

```
curry2( minus, 3, 2 )   // yields a new thunk "f()=3-2"
```

FC++ functoids can be called using a special infix syntax (implemented by overloading `operator^`):

```
x ^f^ y        // Same as f(x,y).  Example: 3 ^plus^ 2
```

This syntax was also inspired by Haskell; some function names (like `plus`) are more readable as infix than as prefix.

FC++ defines *indirect functoids*, which are function variables which can be bound to any function with the same (monomorphic) signature. Indirect functoids are implemented via the `FunN` classes, which take N template arguments describing the argument types, as well as a template argument describing the result type. For example:

```
// Note: plus is polymorphic, the next line selects just "int" version
Fun2<int,int,int> f = plus;
f(3,2);          // yields 5
f = minus;
f(3,2);          // yields 1
```

Indirect functoids are particularly useful in the implementation of callback libraries and some design patterns[11].

The FC++ library defines a number of effect combinators. An effect combinator combines an effect (represented as a thunk) with another functoid. Here are some example effect combinators:

```
// before(thunk,f)(args)  ==  { thunk(); return f(args); }
// after(g,thunk)(args)   ==  { R r = g(args); thunk(); return r; }
```

An example: suppose you've defined a functoid `writeLog()` which takes a string and writes it to a log file. Then

```
before( curry1( writeLog, "About to call foo()" ), foo )
```

results in a new functoid with the same behavior as `foo()`, only it writes a message to the log file before calling `foo()`.

FC++ interfaces with normal C++ code and the STL. The `List` class implements the iterator interface, so that lists can work with STL algorithms and other STL data structures can be converted into `List`s. The functoid `ptr_to_fun()` transforms normal C++ function pointers into functoids, and turns method pointers into functions which take a pointer to the receiver object as an extra first object. Here are some examples, which use currying to demonstrate that the result of `ptr_to_fun` is a functoid:

```
ptr_to_fun( &someFunc )(x)(y)          // someFunc(x,y)
ptr_to_fun( &Foo::meth )(aFooPtr)(x)   // aFooPtr->meth(x)
```

FC++ comes with its own reference-counted smart pointers: `Ref` and `IRef`. `Ref<T>` works just like a `T*`, only with reference counting. `IRef<T>` implements intrusive reference counting; an efficient form of reference counting which requires supportive help from the type being used (here, `T`). Internally, the library uses `IRef`s in the implementation of `List`s and indirect functoids.

4 Where is the magic?

In the previous section we saw how functoids can be used. Nevertheless, we have not shown you how the polymorphic functoids inside FC++ are implemented or how to define your own polymorphic functoids. In this section we show how functoids are defined, and how they gain the special functionality FC++ supports (like currying and infix syntax).

4.1 Defining polymorphic functoids

To create your own polymorphic functoid, you need to create a class with two main elements: a template `operator()` and a member structure template named `Sig`. To make things concrete, consider the definition of `map` (or rather, the class `Map`, of which `map` is a unique instance) shown in Figure 1. This definition uses the helper template `FunType`, which is a specialized template for different numbers of arguments. For two arguments, `FunType` is essentially:

```
struct Map {
   template <class F, class L>
   struct Sig : public FunType<F,L,List<typename F::template
      Sig<typename L::ElementType>::ResultType> > {};

   template <class F, class T>
   typename Sig<F, List<T> >::ResultType
   operator()( const F& f, const List<T>& l ) const {
      if( null(l) )
         return NIL;
      else
         return cons( f(head(l)), curry2(Map(), f, tail(l)) );
   }
} map;
```

Fig. 1. Defining `map` in FC++

```
template <class A1, class A2, class R> struct FunType {
  typedef R ResultType; typedef A1 Arg1Type; typedef A2 Arg2Type;  };
```

We can now analyze the implementation of `Map`. The `operator()` will allow instances of this class to be used with regular function call syntax. What is special in this case is that the operator is a template, which means that it can be used with arguments of multiple types. When an instance of `Map` is used with arguments `f` and `l`, unification will be attempted between the types of `f` and `l`, and the declared types of the parameters (`const F&`, and `const List<T>&`). The unification will yield the values of the type parameters `F` and `T` of the template. This will determine the return type of the functoid.

Now, let's examine the `Sig` member class of the `Map` class. By FC++ convention, the `Sig` member should be a template over the argument types of the function you want to express (in this case the function type `F` and the list type `L`). The `Sig` member template is used to answer the question "what type will your function return if I pass it these argument types?" The answer in the `Map` code is:

```
List< F::Sig<L::ElementType>::ResultType >
```

(we have elided the `typename` and `template` keywords for readability). This means: "map returns a `List` of what `F` would return if passed an element like the ones in list `L`".

In Haskell, one would express the type signature of `map` as:

```
map :: (a -> b) -> [a] -> [b]
```

The `Sig` members of FC++ functoids essentially encode the same information, but in a computational form: `Sig`s are type-computing compile-time functions that are called by the C++ unification mechanism for function templates and implement the FC++ type system. This type system is completely independent

from the native C++ type system—`map`'s type as far as C++ is concerned is just `class Map`. Other FC++ functoids, however, can read the FC++ type information from the `Sig` member of `Map` and use it in their own type computations. The `map` functoid itself uses that information from whatever functoid happens to be passed as its first argument (see the `F::Sig<L::ElementType>::ResultType` expression, above).

4.2 Using the `FullN` wrappers to gain functionality

The definition of `map` in the previous subsection creates what we call a "basic direct functoid" in FC++. However, a number of features of functoids (such as currying and infix syntax, which we saw in Section 3, and lambda-awareness, which will shall describe in Section 5) only work on so-called "full functoids".

Transforming a normal functoid into a full functoid is easy. For example, to define `map` as a full functoid, we change the definition from Figure 1 from

```
struct Map { /* ... */ } map;
```

to

```
struct XMap { /* ... */ };
typedef Full2<XMap> Map;
Map map;
```

That is, `FullN<F>` is the type of the full functoid created out of the basic N-argument functoid `F`. The `FullN` template classes serve as a wrapper around basic functoids. They add all of the FC++ features we are accustomed to (such as currying and infix syntax) to the basic functoid.

Full functoids are a new feature of the FC++ library. Legacy code can promote its basic functoids into full functoids either by making the minor modification to the definition described above, or within an expression by calling the functoid `makeFullN()`, which takes an N-argument basic functoid as an argument and returns the corresponding full functoid as a result.

5 Lambda

Lambda is no stranger to C++. There are a number of existing C++ libraries which enable clients to create new, anonymous functions on-the-fly. Some such libraries, like the C++ STL[12] and its "binders", or previous versions of FC++, allow the creation of new functoids on-the-fly only either by binding some subset of a functions arguments to values (currying) or by using combinators (like `compose`). Other libraries, like the Boost Lambda Library[6] and FACT![9] enable the creation of arbitrary lambdas by using expression templates.

5.1 Motivation

We were motivated to implement lambda by our interest in programming with monads. Experience with previous versions of FC++ made it clear that arbitrary lambdas are a practical necessity if one wants to program with monads. Older versions of FC++ had a number of useful combinators which made it possible to express most arbitrary functions, but lambda makes it practical by making it readable. For example, while implementing a monad, in the middle of an expression you might discover that you need a function with this meaning:

```
lambda(x) { f(g(x),h(x)) }
```

It is possible to implement this function using combinators (without lambda), but the resulting code is practically unreadable:

```
duplicate(compose(flip(compose)(h),compose(f,g)))
```

Alternatively, you can define the new functoid at the top level, give it a name, and then call it:

```
struct XFoo {
   template <class X> struct Sig : public FunType<X,
      typename RT<F<typename RT<G,X>::ResultType,
      typename RT<H,X>::ResultType>::ResultType> {};
   template <class X>
   typename Sig<X>::ResultType operator()( const X& x ) const {
      return f(g(x),h(x));
   }
};
typedef Full1<XFoo> Foo;
Foo foo;
// later use "foo"
```

but clearly this is way too much work, especially when the function in question is a one-time-use ("throwaway") function. Lambda is the only reasonable solution when you need to define short, readable, arbitrary functions on-the-fly.

5.2 Problematic issues with expression-template lambda libraries

Despite the advantages to lambda, we have always maintained a degree of wariness when it comes to C++ lambda libraries (or any expression template library), owing to the intrinsic limitations and caveats of using expression templates in C++. The worrisome issues with expression template libraries in general (or lambda libraries in particular) fall into four major categories:

- **Accidental/early evaluation.** The biggest problem with expression template lambda libraries comes from accidental evaluation of C++ expressions. Consider a short example using the Boost Lambda Library:

  ```
  int a[] = { 5, 3, 8, 4 };
  for_each( a, a+4, cout << _1 << "\n" );
  ```

The third argument to `for_each()` creates an anonymous function to print each element of the array (one element per line). The output is what we would expect:

```
5
3
8
4
```

If we want to add some leading text to each line of output, it is tempting to change the code like this:

```
int a[] = { 5, 3, 8, 4 };
for_each( a, a+4, cout << "Value: " << _1 << "\n" );
```

But (surprise!), the new program prints the added text only once (rather than once per line):

```
Value: 5
3
8
4
```

This is because "`cout << "Value: "`" is a normal C++ expression that the C++ compiler evaluates immediately. Only expressions involving placeholder variables (like `_1`)[1] get "delayed" from evaluation by the expression templates. These accidents are easy to make, and hard to see at a glance.

- **Capture semantics (lambda-specific).** Since C++ is an effect-ful language, it matters whether free variables captured by lambda are captured by-value or by-reference. The library must choose one way or the other, or provide a mechanism by which users can choose explicitly.
- **Compiler error messages.** C++ compilers are notoriously verbose when it comes to reporting errors in template libraries. Things are even worse with expression template libraries, both because there tend to be more levels of depth of template instantiations, and because the expression templates typically expose clients to some new/unfamiliar syntax, which makes it more likely for clients to make accidental errors. Indecipherable error messages may make an otherwise useful library be too annoying for clients to use.
- **Performance.** Expression template libraries sometimes take orders of magnitude longer to compile than comparably-sized C++ programs without expression templates. Also, the generated binary executables are often much larger for programs with expression templates.

For the most part, these problems are intrinsic to all expression template libraries in C++. As a result, when we set out to design a lambda library for FC++, we kept in mind these issues, and tried to design so as to minimize their impact.

[1] Additionally, one can use other special constructs defined by BLL. In the example above, we could get the desired behavior by calling the BLL function `constant()` on the literal string, to delay evaluation.

5.3 Designing for the issues

Here are the design decisions we have made to try to minimize the issues described in the previous subsection.

- **Accidental/early evaluation.** Since the problem itself is intrinsic to the domain, the only way to "attack" this issue is prevention. That is, we cannot prevent users from making mistakes, but we can try to design our lambda to make these mistakes less common and/or more immediately apparent. To this end, we have designed the lambda syntax to be minimalist and visually distinct:
  - **Minimalism.** Rather than overload a large number of operators and include a large number of primitives, we have chosen a minimalist approach. Thus we have only overloaded four operators for lambda language (array brackets for postfix function application, modulus for infix function application, comma for function argument lists, and equality for "let" assignments). Similarly, apart from `lambda`, the only primitives we provide are those for `let`, `letrec`, and if-then-else expressions. These provide a minimal core of expressive power for lambda, without overburdening the user with a wide interface. A narrow interface seems more likely to be remembered and thus less error-prone.
  - **Visual distinctiveness.** Rather than trying to make lambda expressions "blend in" with normal C++ code, we have done the opposite. We have chosen operators which look big and boxy to make lambda expressions "stand out" from normal C++ code. By convention, we name lambda variables with capital letters. By making lambda expressions visually distinct from normal C++ code, we hope to remind the user which code is "lambda" and which code is "normal C++", so that the user won't accidentally mix the two in ways which create accidents of early evaluation.
- **Capture semantics (lambda-specific).** The FC++ library passes arguments by `const&` throughout the library. Effectively this is just another (perhaps efficient) way of saying "by value". As a result, FC++ lambdas capture free variables by value. As with the rest of the FC++ library, the user can explicitly choose reference semantics by capturing *pointers* to objects, rather than the capturing objects themselves.
- **Compiler error messages.** Meta-programming can be used to detect some user errors and diagnose them "within the library" by injecting *custom error messages*[9, 10] into the compiler output. Though many kinds of errors cannot be caught early by the library (lambdas and functoids can often be passed around in potentially legal contexts, but then finally used deep within some template in the wrong context), there are a number of common types of errors that can be nipped in the bud. The FC++ lambda library catches a number of these types of errors and generates custom error messages for them.
- **Performance.** There seems to be little that we (as library authors) can do here. As expression template libraries continue to become more popular, we

can only hope that compilers will become more adept at compiling them quickly. In the meantime, clients of expression template libraries must put up with longer compile times and larger executables.

Thus, given the intrinsic problems/limitations of expression template libraries, we have designed our library to try to minimize those issues whenever possible.

5.4 Lambda in FC++

We now describe what it looks like to do lambda in FC++. Figure 2 shows some examples of lambda. There are a few points which deserve further attention.

```
// declaring lambda variables
LambdaVar<1> X;
LambdaVar<2> Y;
LambdaVar<3> F;

// basic examples
lambda(X,Y)[ minus[Y,X] ]       // flip(minus)
lambda(X)[ minus[X,3] ]         // minus(_,3)

// infix syntax
lambda(X,Y)[ negate[ 3 %multiplies% X ] %plus% Y ]

// let
lambda(X)[ let[ Y == X %plus% 3,
                F == minus[2]
           ].in[ F[Y] ] ]

// if-then-else
lambda(X)[ if0[ X %less% 10, X, 10 ] ]   // also if1, if2

// letrec
lambda(X)[ letrec[ F == lambda(Y)[ if1[ Y %equal% 0,
                                        1,
                                        Y %multiplies% F[Y%minus%1] ]
           ].in[ F[X] ] ]    // factorial
```

Fig. 2. Lambda in FC++

Inside lambda, one uses square brackets instead of round ones for postfix functional call. (This works thanks to the lambda-awareness of full functoids, mentioned in Section 4.) Similarly, the percent sign is used instead of the carat for infix function call. These symbols make lambda code visually distinct so that the appearance of normal-looking (and thus potentially erroneous) code inside a lambda will stand out. Since `operator[]` takes only one argument in C++,

we overload the comma operator to simulate multiple arguments. Occassionally this can cause an early evaluation problem, as seen in the code here:

```
// assume f takes 3 integer arguments
lambda(X)[ f[1,2,X] ]    // oops! comma expression "1,2,X" means "2,X"
lambda(X)[ f[1][2][X] ]  // ok; use currying to avoid the issue
```

Unfortunately, C++ sees the expression "`1,2`" and evaluates it eagerly as a comma expression on integers.[2] Fortunately, there is a simple solution: since all full functoids are curryable, we can use currying to avoid comma. The issues with comma suggest another problem, though: how do we call a zero-argument function inside lambda? We found no pretty solution, and ended up inventing this syntax:

```
// assume g takes no arguments and returns an int
// lambda(X)[ X %plus% g[] ]   // illegal: g[] doesn't parse
lambda(X)[ X %plus% g[_*_] ]   // _*_ means "no argument here"
```

It's better to have an ugly solution than none at all.

The if-then-else construct deserves discussion, as we provide three versions: `if0`, `if1`, and `if2`. `if0` is the typical version, and can be used in most instances. It checks to make sure that its second and third arguments (the "then" branch and the "else" branch) will have the same type when evaluated (and issues a helpful custom error message if they won't). The other two ifs are used for difficult type-inferencing issues that come from `letrec`. In the factorial example at the end of Figure 2, for example, the "else" branch is too difficult for FC++ to predict the type of, owing to the recursive call to `F`. This results in `if0` generating an error. Thus we have `if1` and `if2` to deal with situations like these: `if1` works like `if0`, but just assumes the expression's type will be the same as the type of the "then" part, whereas `if2` assumes the type is that of the "else" part. In the factorial example, `if1` is used, and thus the "then" branch (the `int` value `1`) is used to predict that the type of the whole `if1` expression will be `int`.

Having three different ifs makes the lambda interface a little more complicated, but the alternatives seemed to be either (1) to dispose of custom error messages diagnosing if-then-elses whose branches had different types, or (2) to write meta-programs to solve the recursive type equations created by `letrec` to figure out its type within the library. Option (1) is unattractive because the compiler-generated errors from non-parallel if-then-elses are hideous, and option (2) would greatly complicate the metaprogramming in the library and slow down compile-times even more. Thus we think our design choice is justified. Of course, in the vast majority of cases, `if0` is sufficient and this whole issue is moot; only code which uses `letrec` may need `if1` or `if2`.

5.5 Naming the C++ types of lambda expressions

Expression templates often yield objects with complex type names, and FC++ lambdas are no different. For example, the C++ type of

[2] Some C++ compilers, like g++, will provide a useful warning diagnostic ("left-hand-side of comma expression has no effect"), alerting the user to the problem.

```
// assume:  LambdaVar<1> X;  LambdaVar<2> Y;
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
```

is

```
fcpp::Full2<fcpp::fcpp_lambda::Lambda2<fcpp::fcpp_lambda::exp::
Call<fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Value<
fcpp::Full2<fcpp::impl::XPlus> >,fcpp::fcpp_lambda::exp::CONS<
fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Call<fcpp::
fcpp_lambda::exp::Value<fcpp::Full2<fcpp::impl::XMultiplies> >,
fcpp::fcpp_lambda::exp::CONS<fcpp::fcpp_lambda::exp::Value<int>,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<1>,fcpp::fcpp_lambda::exp::NIL> >,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<2>,fcpp::fcpp_lambda::exp::NIL> >,1,2> >
```

In the vast majority of cases, the user never needs to name the type of a lambda, since usually the lambda is just being passed off to another template function. Occasionally, however, you want to store a lambda in a temporary variable or return it from a function, and in these cases, you'll need to name its type. For those cases, we have designed the `LEType` type computer, which provides a way to name the type of a lambda expression (LE). In the example above, the type of

```
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
// desugared: lambda(X,Y)[ plus[ multiplies[3][X] ][Y] ]
```

is

```
LEType< LAM< LV<1>, LV<2>,
CALL<CALL<Plus,CALL<CALL<Multiplies,int>,LV<1> > >,LV<2> > > >::Type
```

The general idea is that

```
LEType< Translated_LambdaExp >::Type
```

names the type of `LambdaExp`. Each of our primitive constructs in lambda has a corresponding translated version understood by `LEType`:

```
CALL            [] (function call)
LV              LambdaVar
IF0,IF1,IF2     if0[],if1[],if2[]
LAM             lambda()[]
LET             let[].in[]
LETREC          letrec[].in[]
BIND            LambdaVar == value
```

With `LEType`, the task of naming the type of a lambda expression is still onerous, but `LEType` at least makes it possible. Without the `LEType` type computer, the type of lambda expressions could only be named by examining the library implementation, which may change from version to version. `LEType` guarantees a consistent interface for naming the types of lambda expressions.

Finally, it should be noted that if the lambda only needs to be used monomorphically, it is far simpler (though potentially less efficient) to just use an indirect functoid:

```
// Can name the monomorphic "(int,int)->int" functoid type easily:
Fun2<int,int,int> f = lambda(X,Y)[ (3 %multiplies% X) %plus% Y ];
```

5.6 Comparison to other lambda libraries

Here we briefly compare our approach to implementing lambda to that of the other major lambda libraries for C++: the Boost Lambda Library (BLL)[6] and FACT![9].[3]

Boost Lambda Library Whereas FC++ takes the minimalist approach, BLL takes the maximal approach. Practically every overloadable operator is supported within lambda expressions, and the library has special lambda-expression constructs which mimic the control constructs of C++ (like while loops, switches, exception handling, etc). The library also supports making references to variables, and side-effecting operators like increment and assignment. Lambda is implicit rather than explicit; a reference to a placeholder variables (like `_1`) turns an expression into a lambda on-the-fly.

BLL's approach makes sense given the "target audience"; the Boost libraries are designed for everyday C++ programmers. These are people who are familiar with C++ constructs, and who are hopefully C++-savvy enough to avoid most of the pitfalls of an expression-template lambda library. In contrast, FC++ is designed to support functional programming in the style of languages like Haskell. A number of our users come from other-language backgrounds, and aren't too familiar with the intricacies of C++. Thus FC++'s lambda is designed to present a simple interface with syntax and constructs familiar to functional programmers, and to shield users from C++-complexities as much as possible.

FACT! FACT!, like FC++, is designed to support pure functional programming constructs. Lambda expressions always perform capture "by value" and the resulting functions are typically effect-free. Like FC++, FACT! has an explicit lambda construct; the user can define his own names for placeholder variables, but conventionally names like `x` and `y` are used. FACT! defines few primitive control constructs in its lambda sublanguage (just `where` for if-then-else). Like BLL, however, FACT! overloads many C++ operators (like `+`) for use in lambda expressions. Thus FACT!'s interface is relatively simple and minimal, but lambda expressions are not as visually distinctive as they are in FC++.

6 Monads

Monads provide a useful way to structure programs in a pure functional language. Using monads, it is relatively straightforward to implement things like global

[3] The FACT! library, like FC++, includes features other than lambda, e.g. functions like `map()` and `foldl()` as well as data structures for lazy evaluation. BLL, on the other hand, is concerned only with lambda.

state, exceptions, I/O, and other concepts common to impure languages that are otherwise difficult to implement in pure functional languages[6, 14].

Supporting monads in FC++ is an interesting task for a number of reasons:

- Many interesting functional programs and libraries use monads; monad support in FC++ makes it easier to port these libraries to C++.
- Monads in Haskell take advantage of some of that language's most expressively powerful syntax and constructs, including *type classes*, *do-notation*, and *comprehensions*. Modelling these in C++ helps us better understand the relationship between the expressive power of these languages.
- Monads provide a way to factor out some cross-cutting concerns, so that local program changes can have global effects. (We discuss a few example applications that illustrate this.)

In the next subsection, we give a short introduction to monadic programming in Haskell. Next we discuss the relationship between *type classes* in Haskell and *concepts* in C++; understanding this relationship facilitates the discussion in the rest of this section. Then we discuss how we have implemented monads in FC++. We end with some example applications of monads.

6.1 Introduction to monads in Haskell

We briefly introduce a small portion of the Haskell programming language,[4] as its type system provides perhaps the most succinct and transparent way to understand the details of what a monad is. For the moment, know that a monad is a particular kind of data type, which supports two operations (named `unit` and `bind`) with certain signatures that obey certain properties. We shall return to the details after a short digression with Haskell.

In Haskell, the declaration `o :: T` says that object `o` has type `T`. Basic type names (like `Int`) start with capital letters. Lowercase letters are used for free type variables (parametric polymorphism – e.g. templates). The symbol `[T]` represents a list of `T` objects. The symbol `->` separates function arguments and results. The symbol `--` starts a comment. Here are a few examples.

```
x    :: Int                 -- x is an integer

add1 :: Int -> Int          -- add1 is a function from Int to Int

plus :: Int -> Int -> Int   -- plus takes two Ints and returns an Int
   -- (Or, equivalently, plus takes one Int, and returns a function
   --  which takes an Int and returns an Int.  Currying is built in.)

id   :: a -> a              -- id takes any type of object and returns
                            -- an object of the same type

map  :: (a -> b) -> [a] -> [b] -- map is a polymorphic function of two
```

[4] Haskell programmers will note that we are fudging some of the details of Haskell to simplify the discussion.

```
   -- arguments; it takes a function from type a to type b, and a
   -- list of objects of type a, and returns a list of b objects
```

Free type variables can be bounded by "type classes" (described shortly). For example, a function to sort a list requires that the type of elements in the list are comparable with the less-than operator. In Haskell we would say:

```
sort :: (Ord a) => [a] -> [a]
```

That is, `sort` is a function which takes a list of `a` objects and returns a list of `a` objects, subject to the constraint that the type `a` is a member of the `Ord` type class. Type class `Ord` in Haskell represents those types which support ordering operators like

```
class Ord a where
   ==  :: a -> a -> Bool
   <   :: a -> a -> Bool
   <=  :: a -> a -> Bool
      -- etc.
```

We say that a type `T` is an *instance* of type class `C` when the type supports the methods in the type class. For example, it is true that

```
instance Ord Int     -- Int is an instance of Ord
```

Given this overview of Haskell's types and type classes, we can now describe monads. A monad is a type class with two operations:

```
class Monad m where
   bind :: m a -> ( a -> m b ) -> m b
   unit :: a -> m a
```

In this case, instances of monads are not types, but rather they are "type constructors". These are like template classes in C++; an example is a list. In C++ `std::list` is not a type, but `std::list<int>` is. The same holds for Haskell; `[]` is not a type, but `[Int]` is. In the code describing the monad type class above, `m` is a type constructor.

It turns out that *lists* are instances of monads:

```
instance  Monad []  where
    bind m k       = concat (map k m)   -- don't worry about these
    unit x         = [x]                -- definitions yet
-- in the list monad
-- bind :: [a] -> ( a -> [b] ) -> [b]
-- unit :: a -> [a]
```

As another example, consider the `Maybe` type constructor. The type "`Maybe a`" represents a value which is either just an `a` object, or else nothing. In Haskell:

```
data  Maybe a  =  Nothing | Just a

-- Examples of variables
x  :: Maybe Int
```

```
x  =  Just 3

y  :: Maybe Int
y  =  Nothing
```

`Maybe` also forms a monad with this definition:

```
instance  Monad Maybe  where
    bind (Just x) k   =  k x            -- don't worry about
    bind Nothing  k   =  Nothing        -- these definitions
    unit x            =  Just x         -- yet
-- in the Maybe monad
-- bind :: Maybe a -> ( a -> Maybe b ) -> Maybe b
-- unit :: a -> Maybe a
```

A refinement of the `Monad` type class is `MonadWithZero`:

```
class (Monad m) => MonadWithZero m where
   zero :: m a
```

The `zero` element of a monad is a value which is in the monad regardless of what type was passed to the monad type constructor. For lists, the empty list (`[]`) is the `zero`. For `Maybe`, the `zero` is `Nothing`. Not all monads have zeroes, which is why `MonadWithZero` is a separate type class.

Monads with zeroes can be used in *comprehensions* with *guards*. Comprehensions are a special notation for expressing computations in a monad. Haskell supports comprehensions for the list monad; an example is

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
-- results in [3,4,5]
```

This list comprehension could be interpreted as "the list of values x plus y, for all x and y where x is selected from the list [1,2,3] and y is selected from the list [2,3], and where x is less than y". The desugared version of the Haskell code is:

```
-- (\z -> z+1) is Haskell lambda syntax:  (lambda(Z)[ Z %plus% 1 ])
-- backquotes are Haskell's infix syntax: (x `f` y == f x y)
[1,2,3] `bind` (\x ->
   [2,3]   `bind` (\y ->
      if not (x<y) then zero
                   else unit (x+y) ))
```

The translation from the comprehension notation to the desugared code is straightforward. Starting from the vertical bar and going to the right, the expressions of the form "`var <- exp`" turn into calls to `bind` and lambdas, and guards (boolean conditions) are transformed into if-then-else expressions which return the monad `zero` if the condition fails to hold. After all expressions to the right of the vertical bar have been processed, the expression to the left of the vertical bar gets `unit` called on it to lift the final computed value back into the monad.

6.2 Haskell's type classes and C++ template concepts

In the C++ literature, we sometimes speak of template *concepts*. A concept in C++ is a set of constraints which a type is required to meet in order to be used to instantiate a template. For example, in the implementation of the template function `std::find()`, there will undoubtedly be some code along the lines of

```
if( cur_element == target ) // ...
```

which compares two elements for equality using the equality operator. Thus, in order to call `std::find()` to find a value in a container, the element type must be `EqualityComparable`—that is, it must support the equality operator with the right semantics. We call `EqualityComparable` a *concept*, and we say that types (such as `int`) which meet the constraints *model* the concept. Concepts exist only implicitly in the C++ code (e.g. owing to the call to `operator==()` in the implementation), and often exist explicitly in documentation of the library. Some C++ libraries[9, 10] are devoted to "concept checking", these libraries check to see that the types used to instantiate a template do indeed model the required concepts (and issue a useful error message if not).

Haskell type classes are analogous to C++ concepts. However in Haskell they are reified; there are language constructs to define type classes and to declare which types are instances of those type classes. In C++, when a certain type models a certain concept (by meeting all of the appropriate constaints), it is merely happenstance (structural conformance); in Haskell, however, in addition to meeting the constraints of a type class interface, a type must be declared to be an instance of the concept (named conformance). "Concept checking" in Haskell is built into the language: type classes define concepts, instance declarations say which types model which concepts, and type bounds make explicit the constraints on any particular polymorphic function.

Understanding this analogy will make the FC++ implementation of monads more transparent. As we shall see, in the FC++ library, we spell out the concept requirements on monads, in order to make it easier for clients who write monads to ensure that they have provided all of the necessary functionality in the templates.

6.3 Comparing monads in FC++ to those in Haskell

Let us now illustrate monad definitions in FC++. As a first example, we shall look at `Maybe`. The `Maybe` template class and its associated entities are defined in Figure 3. `NOTHING` is the constant which represents an "empty" `Maybe`, and `just()` is a functoid which turns a value of type `T` into a "full" `Maybe<T>`. (`Maybe` is implemented using a `List` which holds either one or zero elements.)

Next we consider how to make `Maybe` a monad. Figure 4 describes the general monad concepts as specified in the FC++ documentation. A monad class must define the methods `unit` and `bind` (with the appropriate signatures); a class representing a monad with a zero must meet the above requirements as well as defining a `zero` element.

```
struct AUniqueTypeForNothing {};
AUniqueTypeForNothing NOTHING;

template <class T>
class Maybe {
   List<T> rep;
public:
   typedef T ElementType;

   Maybe( AUniqueTypeForNothing ) {}
   Maybe() {}                                    // Nothing constructor
   Maybe( const T& x ) : rep( cons(x,NIL) ) {}  // Just constructor

   bool is_nothing() const { return null(rep); }
   T value() const { return head(rep); }
};

struct XJust {
   template <class T> struct Sig : public FunType<T,Maybe<T> > {};

   template <class T>
   typename Sig<T>::ResultType
   operator()( const T& x ) const {
      return Maybe<T>( x );
   }
};
typedef Full1<XJust> Just;
Just just;
```

Fig. 3. The `Maybe` datatype in FC++

```
/*
   concept Monad {
      // full functoid with Sig   unit :: a -> m a
      typedef Unit;
      static Unit unit;
      // full functoid with Sig   bind :: m a -> ( a -> m b ) -> m b
      typedef Bind;
      static Bind bind;
   }
   concept MonadWithZero models Monad {
      // zero :: m a
      typedef Zero;           // a value type
      static Zero zero;
   }
*/
```

Fig. 4. Documentation of the monad concept requirements in FC++

```
struct MaybeM {
   typedef Just Unit;
   static Unit unit;

   struct XBind {
      template <class M, class K> struct Sig : public FunType<M,K,
         typename RT<K,typename M::ElementType>::ResultType> {};
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const {
         if( m.is_nothing() )
            return NOTHING;
         else
            return k( m.value() );
      }
   };
   typedef Full2<XBind> Bind;
   static Bind bind;

   typedef AUniqueTypeForNothing Zero;
   static Zero zero;
};
```

Fig. 5. Definition of the `Maybe` monad (`MaybeM`)

Figure 5 shows how we define the `Maybe` monad in FC++. Nested in `struct MaybeM` we define `unit`, `bind`, and `zero`, as well as `typedef`s for their types. This FC++ definition effectively corresponds to the definitions

```
instance Monad Maybe -- ...
instance MonadWithZero Maybe -- ...
```

in Haskell.

It should be noted here that the one major difference between monads in FC++ and monads in Haskell is that, in FC++, there is a distinction between the monad type constructor (e.g. `Maybe`) and the monad itself (e.g. `MaybeM`). We chose to make this distinction for reasons discussed next.

One advantage to separating the type constructor (`Maybe`) from the monad definition (`MaybeM`) is that, since the monad definition is itself a data type, it can be used as a type parameter to template functions. As a result, rather than supporting just list comprehensions (like Haskell does), in FC++ we support *comprehensions in an arbitrary monad*, by passing the monad as a template parameter to the comprehension. For example, the Haskell list comprehension

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
```

is written in FC++ as

```
compM<ListM>()[ X %plus% Y |
   X <= list_with(1,2,3), Y <= list_with(2,3), guard[ X %less% Y ] ]
```

Note how `ListM` is passed as an explicit template parameter to the `compM` function, which returns a comprehension for that monad. As a result, we can write

```
compM<MaybeM>()[ X %plus% Y | X <= just(2), Y <= just(3) ]
```

and perform a comprehension in the `Maybe` monad. Having a name apart from the data type constructor to serve as a handle for the monad definition (e.g. `ListM`, `MaybeM`) gives us a convenient way to parameterize monad operations. (The idea of generalizing comprehensions to arbitrary monads was originally discussed by Wadler[15].)

There is another advantage to separating the type constructor from the monad definition. Haskell type classes require algebraic data type constructors (not type aliases) to work. As a result, we cannot express the identity monad (a monad where `m a = a`) directly in Haskell. Instead we have to fake it by defining a new data type (which we have chosen to call `Identity`):

```
data Identity a = Ident a

instance Monad Identity where    --  m a = Identity a
   unit x   = x
   bind m k = k m
```

where values of type `a` are wrapped/unwrapped with the value constructor `Ident` to make them members of the type `Identity a`. In FC++, however, we can define the monad without also having to define a new data type to represent identities, as seen in Figure 6. The reason for the distinction is perhaps obvious. Haskell uses type inference, which means it must unambiguously be able to figure out which monad a particular data type is in. This type inference is not possible unless there is a one-to-one mapping between algebraic datatype constructors and monads. In FC++, on the other hand, the user passes the monad explicitly as a template parameter to constructs like `compM`. By requiring the user to be a little more explicit about the types, we gain a bit of expressive freedom (e.g. being able to do comprehensions in arbitrary monads).

6.4 Monads in FC++

The previous subsection introduced FC++ monads. Here we flesh out exactly what monad support FC++ provides.

FC++ provides functoids for the main monad operations. Specifically:

```
unitM<SomeMonad>()    // SomeMonad's "unit" functoid
bindM<SomeMonad>()    // SomeMonad's "bind" functoid
zeroM<SomeMonad>()    // SomeMonad's "zero" value
plusM<SomeMonad>()    // SomeMonad's "plus" functoid
bindM_<SomeMonad>()   // SomeMonad's "bind_" functoid
mapM<SomeMonad>()     // SomeMonad's "map" functoid
joinM<SomeMonad>()    // SomeMonad's "join" functoid
liftM<SomeMonad>()    // lifts a one-arg function into SomeMonad
liftM2<SomeMonad>()   // lifts a two-arg function into SomeMonad
```

```
// Nothing corresponding to Identity data type needed by Haskell
struct IdentityM {    // M a = a
   typedef Id Unit;
   static Unit unit;

   struct XBind {
      template <class M, class K> struct Sig : public FunType<M,K,
         typename RT<K,M>::ResultType> {};
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const {
         return k(m);
      }
   };
   typedef Full2<XBind> Bind;
   static Bind bind;
};
```

Fig. 6. Definition of the `IdentityM` monad

```
liftM3<SomeMonad>()   // lifts a three-arg function into SomeMonad
bind                  // "bind" (monad is inferred)
bind_                 // "bind_" (monad is inferred)
```

Many of these have not been previously mentioned; `plusM` is another function supported by some monads; `bindM_`, `mapM`, `joinM`, and the `liftM` functions are common monad operations which are defined in terms of `unitM` and `bindM`; `bind` and `bind_` are described more below.

FC++ supports comprehensions in arbitrary monads, using the general syntax:

```
compM<SomeMonad>()[ lambdaExp | thing, thing, ... thing ]
```

where `thing` is one of

- a gets expression of the form "`LV <= lambdaExp`" (Translates into a call to `bind`)
- a lambda expression (Translates into a call to `bind_`)
- a guard expression of the form "`guard[boolLambdaExp]`" (Translates into an if-then-else with `zero` if the test fails)

This is similar to the syntax used by Haskell's list comprehensions. FC++ also supports a construct similar to Haskell's *do-notation*:

```
doM[ thing, thing, ... thing ]
```

where each `thing` is as before, only `guard`s are no longer allowed. (The lack of a monad parameter to `doM` is discussed shortly.)

Clients can define monads by creating monad classes which model the monad concepts described in the previous subsection (`Monad` and `MonadWithZero`).

There is also a `MonadWithPlus` concept for monads which support `plus`. Additionally there is another concept called `InferrableMonad`, which may be modelled when there is a one-to-one correspondence between a datatype and a monad. In the case of `InferrableMonad`s, FC++ (like Haskell) can automatically infer the monad based on the datatype in some cases; constructs like `doM` and the functoids `bind` and `bind_` do not need to have a monad passed an an explicit parameter—they infer it automatically.

The monad syntax is part of FC++'s lambda sublanguage. As with `lambda`, we strived for minimalism when implementing monads. The only new operator overloads are `operator|` and `operator<=`, and the only new syntax primitives are `compM`, `guard`, and `doM`. As with the rest of `lambda`, we provide `LEType` translations so that clients can name the result type of lambda expressions which use monads:

```
DOM             doM[]
GETS            LambdaVar <= value
GUARD           guard[]
COMP            compM<SomeMonad>()[]
```

As with the other portions of `lambda`, FC++ provides some custom error messages for common abuses of the monad constructs. We followed the same design principles discussed in Section 5 when implementing monads in FC++.

6.5 Monad examples

There are many example applications which use monads; here we discuss a small sample to give a feel for what monads are useful for.

Using MaybeM for exceptions One classic example of the utility of monads comes from the domain of exception handling. Suppose we have written some code which computes some values using some functions:

```
x = f(3);
y = g(x);
z = h(x,y);
return z;
```

(For the sake of argument, let's say that the functions `f`, `g`, and `h` take positive integers as arguments and return positive integers as results.) Now suppose that each of the functions above may fail for some reason. In a language with exceptions, we could throw exceptions in the case of failure. However in a language without an exception mechanism (like C or Haskell), we would typically be forced to represent failure using some sentinel value (`-1`, say), and then change the client code to

```
x = f(3);
if( x == -1 ) {
   return -1;
} else {
```

```
    y = g(x);
    if( y == -1 ) {
       return -1;
    } else {
       z = h(x,y);
       return z;
    }
}
```

This is painful because the "exception handling" part of the code clutters up the main line code. However, we can solve the problem much more simply by using the Maybe monad. Let the functions return values of type `Maybe<int>`, and let `NOTHING` represent failure. Now the client code can be written as just

```
compM<MaybeM>()[ Z | X <= f[3],
                     Y <= g[X],
                     Z <= h[X,Y] ]
```

The definitions of `unit` and `bind` in the `MaybeM` monad make the problem trivial; `NOTHING` values immediately propogate up through the end of the comprehension, whereas integers continue on through the computation as desired.

Using ListM for non-determinism Now imagine changing the problem above slightly; instead of the functions `f`, `g`, and `h` having the possibility of failure, suppose instead that they are non-deterministic. That is, suppose each function returns not a single integer, but rather a list of all possible integer results. Changing the original client code to deal with this change would likely be even uglier than the original change (which required all the tests for `-1`). However the change to the monadic version is trivial:

```
compM<ListM>()[ Z | X <= f[3],    -- Note ListM instead of MaybeM
                    Y <= g[X],
                    Z <= h[X,Y] ]
```

The result is a list of all the possible integer values for `Z` which satistfy the formulae.

A monadic evaluator Wadler [15] demonstrates the utility of monads in the context of writing an expression evaluator. Wadler gives an example of an interpreter for a tiny expression language, and shows how adding various kinds of functionality, such as error handling, counting the number of reduction operations performed, keeping an execution trace, etc. takes a bit of work. The evaluator is then rewritten using monads, and the various additions are revisited. In the monadic version, the changes necessary to effect each of the additions are much smaller and more local than the changes to the original (non-monadic) program. This example demonstrates the value of using monads to structure programs in order to localize the changes necessary to make a wide variety of additions throughout a program.

Monadic parser combinators Parsing is a domain which is especially well-suited to monads. In the Haskell community, "monadic parser combinators" are becoming the standard way to structure parsing libraries. As it turns out, parsers can be expressed as a monad: a typical representation type for parser monads is

```
Parser a = String -> Maybe ( a, String )   -- the monad "Parser"
```

That is, a parser is a function which takes a `String` and returns

- (if the parse succeeds) a pair containing the result of the parse and the remaining (yet unparsed) `String`, or
- (if the parse fails) `Nothing`.

Monadic parser *combinators* are functions which combine parsers to yield new parsers, typically in ways commonly found in the domain of parsing and grammars. For example, the parser combinator `many`:

```
many :: Parser a -> Parser [a]
```

implements Kleene star—for example, given a parser which parses a single digit called "`digit`", the parser "`many digit`" parses any number of digits. Monadic parser combinator libraries typically provide a number of basic parsers (e.g. `charP`, which parses any character and returns that character) and combinators (e.g. `plusP`, which takes two parsers and returns a new parser which tries to parse a string with the first parser, but if that fails, uses the second) to clients. The beauty of the monadic parser combinator approach is that it is easy for clients to define their own parsers and combinators for their specific needs. A good introductory paper on the topic of monadic parser combinators in Haskell is [3]; we implement the examples in that paper in one of the example files that comes with the FC++ library.

As we have seen in the previous examples, using monads often makes it easy to change some fundamental aspect of the behavior of the program. For example, if we have an ambiguous grammar (one for which some strings admit multiple parses), we can simply change the representation type for the parser like so:

```
Parser a = String -> [ ( a, String ) ]  -- uses List instead of Maybe
```

and redefine the monad operations (`unit`, `bind`, `zero`, and `plus`), and then parsers will return a list of every possible parse of the string. This is all possible without making any changes to existing client code.

One alternative approach to writing parsing libraries in C++ is that taken by the Boost Spirit Library[1]. Spirit uses expression templates to turn C++ into a `yacc`-like tool, where parsers can be expressed using syntax similar to the language grammar. For example, given the expression language

```
factor      ::= integer | group              // BNF
term        ::= factor (mulOp factor)*
expression  ::= term (addOp term)*
group       ::= '(' expression ')'
```

one can write a parser using Spirit as

```
factor      = integer | group;                // Spirit (C++)
term        = factor >> *(mulOp >> factor);
expression  = term >> *(addOp >> term);
group       = '(' >> expression >> ')';
```

which is almost just as readable as the grammar. Like `yacc`, Spirit has a way to associate semantic actions with each rule.

The results are similar with monadic parser combinators. Using an FC++ monadic parser combinator library, we can write

```
factor     = lambda(S)[ (integer %plusP% dereference[&group])[S] ];
term       = factor ^chainl1^ mulOp;
expression = term ^chainl1^ addOp;
group      = bracket( charP('('), expression, charP(')') );
```

to express the same parser. The above FC++ code creates parser functoids by using more primitive parsers and combining them with appropriate parser combinators like `chainl1`. (Note that, whereas Spirit's parser rules are effectively "by reference", FC++ functoids are "by value", which means we need to explicitly create indirection to break the recursion among these functoids. Hence the use of `lambda`, `dereference`, and the address-of operator.) This FC++ parser not only parses the string, but it also evaluates the arithmetic expression parsed. The semantics are built into the user-defined combinators like `addOp` and `chainl1`. For example,

```
addOp :: Parser (Int -> Int -> Int)
```

parses a symbol like `'-'` and returns the corresponding functoid (`minus`). Then,

```
chainl1 :: Parser a -> Parser (a -> a -> a ) -> Parser a
-- e.g.   p `chainl1` op
```

parses repeated applications of parser `p` , separated by applications of parser `op` (whose result is a left-assocative function, which is used to combine the results from the `p` parsers). Thus monadic parser combinator libraries allow one to express parsers at a level of abstraction comparable to tools like `yacc` or the Spirit library, but in a way in which users can define their own abstractions (like `chainl1`) for parsing and semantics, rather than just using the builtin ones (like Kleene star) supplied by the tool/library.

Lazy evaluation Previous versions of FC++ supported lazy evaluation in two main ways: first, via the lazy `List` class and the functions (like `map`) that use `List`s, and second, via "thunks" (zero argument functoids, like `Fun0<T>`). Monads provide a new, more general mechanism to lazify computations. The datatype `ByNeed<T>` and its associated monad `ByNeedM` can be used to make a computation lazy. Additionally, the functoid `bLift` lazifies a functoid by lifting its result into the ByNeedM monad. For example, we can lazify

```
x = f(3);
y = g(x);
z = h(x,y);
```

by writing

```
compM<ByNeedM>()[ Z | X <= bLift[f] [3],
                      Y <= bLift[g] [X],
                      Z <= bLift[h] [X,Y] ]
```

The result is a `ByNeed<int>` value, which is a computation that will result in an `int` when "forced" by calling `bForce`. (Conversely, a constant can be turned into a by-need computation by calling `bDelay`.) Using values of type `ByNeed<T>` in lieu of type `T` ensures that lazy evaluation occurs: a computation is not performed until the value is demanded, and once a computation has been run to produce a value, the value is cached so that further applications of `bForce` get the cached value rather than re-running the computation.

In short, the datatype `ByNeed<T>` combines "thunks" with caching, and the `ByNeedM` monad makes syntax sugar like comprehensions available so that client code working with `ByNeed<T>` objects need not be concerned with all the "forcing" and "delaying" in the midst of the computation (the monad plumbing handles this).

Summary The examples given in this section give a sense of the kinds of applications for which monads are useful. Monads have a wide variety of utilities, which span varied domains (such as parsing and lists) and a number of cross-cutting concerns (like lazy evaluation and exception handling). Prior versions of FC++ implemented a few small monads, but they were extremely burdensome to express. The expressiveness afforded by the new FC++ syntactic sugar (like lambda and comprehensions) makes using monads in C++ a practicality for the first time.

7 Conclusions

We have given an overview of FC++ and described its new features in detail. Full functoids provide a general and reusable mechanism for adding features such as curryability, infix syntax, and lambda-awareness to every functoid. The lambda sublanguage is designed to minimize the problems common to all expression-template lambda libraries in C++. We have discussed the relationship between Haskell type classes and C++ template concepts in order to help describe how monads can be expressed in FC++. We have demonstrated a novel syntax for comprehensions which generalizes this construct to an arbitrary monad. Throughout FC++ and the lambda sublanguage, we have overloaded a select few operators to provide syntactic sugar for the library and we have used named functoids like `plus` to express the actual operations of C++ operators.

References

[1] de Guzman, Joel, et al. The Boost Spirit Library. Available at `http://www.boost.org/libs/spirit/index.html`

[2] *Haskell 98 Language Report*. Available online at
`http://www.haskell.org/onlinereport/`

[3] Hutton Graham and Meijer Erik. "Monadic parsing in Haskell" *Journal of Functional Programming*, 8(4):437-444, Cambridge University Press, July 1998.

[4] *ISO/IEC 14882: Programming Languages – C++*. ANSI, 1998.

[5] Järvi, Jaakko and Powell, Gary. The Boost Lambda Library. Available at
`http://boost.org/libs/lambda/doc/index.html`

[6] Jones, Simon Peyton and Wadler, Philip. "Imperative functional programming," *20th Symposium on Principles of Programming Languages*, ACM Press, Charlotte, North Carolina, January 1993.

[7] McNamara, Brian and Smaragdakis, Yannis. "FC++: Functional Programming in C++", *Proc. International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.

[8] McNamara, Brian and Smaragdakis, Yannis. "Functional Programming with the FC++ library" *Journal of Functional Programming*, to appear.

[9] McNamara, Brian and Smaragdakis, Yannis. "Static Interfaces in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at
`http://www.oonumerics.org/tmpw00/`

[10] Siek, Jeremy and Lumsdaine, Andrew. "Concept Checking: Binding Parametric Polymorphism in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`

[11] Y. Smaragdakis and B. McNamara, "FC++: Functional Tools for Object-Oriented Tasks" *Software Practice and Experience*, August 2002.

[12] A. Stepanov and M. Lee, "The Standard Template Library", 1995. Incorporated in ANSI/ISO Committee C++ Standard.

[13] Striegnitz, Jörg. "FACT! The Functional Side of C++," Available at
`http://www.fz-juelich.de/zam/FACT`

[14] Wadler, Philip. "Comprehending monads," *Mathematical Structures in Computer Science*, Special issue of selected papers from 6th Conference on Lisp and Functional Programming, 2:461-493, 1992.

[15] Wadler, Philip. "Monads for functional programming." J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.

Importing alternative paradigms into modern object-oriented languages.

Andrey V. Stolyarov

Moscow State Lomonosov University,
dept. of Computational Math. and Cybernetics,
MGU, II uch. korp., komn.747, Leninskie Gory,
Moscow, 119899, Russia

Abstract. The paper is devoted to the problem of importing alternative paradigms into an imperative object-orented environment. Several known solutions of the problem are discussed with explanation of their drawbacks. Then a new solution is introduced.
The solution is based on the fact that programming paradigms developed within alternative languages such as Lisp, Prolog, Refal etc. are in fact independent from their respective languages (e.g., from their syntax). Each of these languages implements a certain algebra, which in fact creates the paradigms. It is possible to represent such an algebra with object-oriented technique and get the respective set of paradigms within the primary language. Together with syntactic capabilities of the primary language (such as overloading of standard arythmetic operation symbols) this results in possibility for a programmer to use alternative paradigms (such as Lisp programming) right within the primary language (C++ or Ada95). No changes to the primary language is needed, nor is it required to apply any additional preprocessing to the code; only the standard translator of the primary language is used. The only thing needed to use the explained approach is an appropriate library.
As an illustration, the paper describes a C++ class library named InteLib which currently has a practically usable implementation for Lisp programming, and experimental umplementations of Refal and a subset of Prolog.

1 Introduction

Different programming languages encourage a programmer to use different ways to imagine the program being developed, the environment (hardware, operating system, user etc) and their interaction. The simplest way is to imagine a computer either as such (processor, memory, i/o ports etc) or a some kind of virtual machine capable to perform certain set of operations, and a program as a sequence of instructions those explicitly specify what operations are to be performed. This way of thinking is known as "imperative programming".

Another technique, proposed in early 1960s [10], is to represent the program as a set of functions. Each function gets zero or more arguments and computes a result. A function may use other functions in computations, including using

itself, directly or indirectly (so called recursive function calls). No side effects are allowed, that is, if all the arguments are known, one can replace a function call with it's result and get the same program. This is known as "pure functional programming" [7].

Several years later, "logic programming" [12] was proposed. In logic programming, the program is thought as a set of logic facts (axioms) used to test statements and find objects that satisfy the given conditions. Pure logic programming also disallows side effects.

Obviously, both pure functional and pure logic programming are not suitable for interactive programs because interactive program changes its environment at least reading from the input and writing to the output so it's required to have functions with side effects to create an interactive program. In contrast with these two styles, the technique of object-oriented programming appeared in the middle 1970s looks like being created specially for interactive software development. The program and its environment are represented with so-called "objects" – abstract "black boxes" capable to exchange messages and perform various actions in response to a message [2].

The notion of a *programming paradigm* is often used to refer to a particular system of programming abstractions. It is important to notice that the notion of a programming paradigm is significantly unformal. There is no well-established classification of programming paradigms though there were many attempts to give one (e.g., [13]). For example, the languages Lisp[15], Refal[17], Miranda[18] and Hope[7] are all usually taken as functional languages. However, they in fact have less in common than in differences. E.g., a Refal function is based on text matching against patterns and transforming in accordance to the rule for the pattern first matched. There's no such capability in Lisp. This makes it more convenient to perform lexical and syntactic analyses with Refal than with Lisp. Hope and Miranda are pure functional languages while Lisp has global variables and local lexical bindings changeable as a side effect of a function, etc.

Nowadays the union of imperative and object-oriented paradigms is the most popular in the software industry. The union is implemented by such languages as C++ [16], Ada95, Java, Delphi, Object Pascal etc.

Using different languages together within a single project leads to different serious problems so it is rare practice; most projects are single-language, and the langage is one of these imperative object-oriented languages listed above. In most cases it is inconvenient to implement a whole project in Lisp, Refal or Prolog. This in fact results in that these languages are rarely used at all despite that they are extremely suitable for some subtasks in almost any project.

2 Example of a project suitable for multiparadigm technique

As noted in [6], "We may encounter application domains which can be modeled best with only one paradigm. But there may be other domains which can be represented more adequately using multiple paradigms". Furthermore, in almost

any large software project there are subtasks for which alternative paradigms are suitable.

Consider we have a database with complicated relationship within its components, and a user who needs a good interface to the database, preferably close to a natural language. First of all, we have to analyse the queries user types at his console (lexical analysis). Then we need to determine what do they mean (syntax analysis). Finally, we need to create and perform a query to the database and return its results to the user.

It takes significant time to write a lexical analyser in an imperative language such as C++. If we do it in Refal, however, the work's complexity reduces by tens of times.

Next, we need to do some more processing to prepare the query. We should determine what formulae do mean and probably perform some transformations, optimizations etc. It is hard to operate with symbolic formulae in C++, but in Lisp all the symbolic transformations are programmed simply.

Now we are ready to request and retrieve a result from the database. It's not a problem when the query is simple; for instance, if we've got a database storing personal data of some people, a request like "Give me a home phone number of Bob Johnson" after lexical analysis and syntax transformations is not a problem to perform. However, the user might ask for a thing much harder to calculate. For example: "Find me a female who has graduated in 1997 in the Moscow State University as a computer science person, then got married in 1998 with a male who speaks fluent English and is older than his wife by 3 years". It is possible to create a database that stores all the necessary data, but a request like this might force us to write a whole program to complete it. Please note that we know the conditions and the only problem is to find the solution which satisfies them all. Logic languages such as Prolog, Datalog etc. are suitable for this purpose [5].

It looks like a good idea to make data flow such as shown at fig. 1

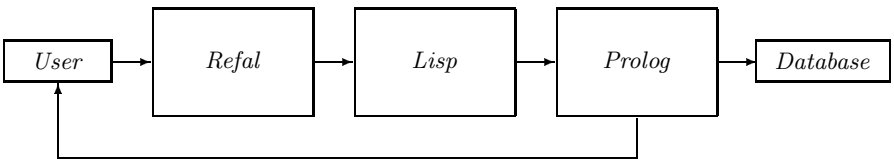

Fig. 1. Simple idea of data flow between parts implemented in different languages

However, the diagram at fig. 1 does not represent all the functionality of the hypotetical system. First, it must interact with a user, possibly via a network. Second, it should control the database, which is probably to be used by many users simultaneously. That leads to many technological problems (e.g., locking) to be solved by the system. In addition, we should not forget that the data is stored on physical (that is, *real*) disks, so we need to monitor the file system,

check whether there's sufficient amount of free space, do some caching to increase performance etc.

Languages such as Refal, Prolog and Lisp are not good to do all these things. Their features are far from the real equipment capabilities. They are not so efficient as C++ and other "universal" imperative languages. For an artificial intelligence tasks such as the one described below, it is possible to trade efficiency of the software for development speed increase, but in a system task envolved to maintain the data storage it is inacceptable to loose in efficiency.

Besides that, interaction with a user in modern systems requires graphical interface (GUI), which is usually created as an event-driven system. It is inconvenient to create an event-driven system with an artificial intelligence language. Object-oriented languages such as SmallTalk[8], C++ or Java are much more suitable for this purpose.

The diagram at fig. 2 is closer to practice. We assume that "User Interface" and "Database Management System" are created with languages suitable for these purposes, probably C and C++. So we have C++ as a primary language and Lisp, Prolog and Refal as secondary languages.

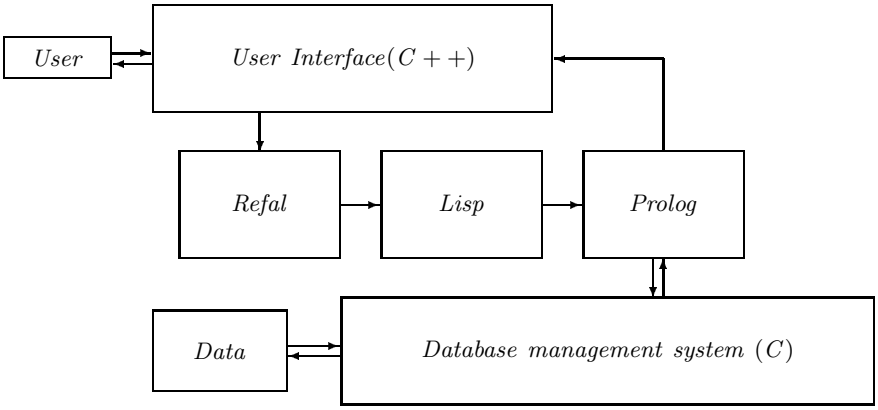

Fig. 2. Data flow closer to reality

However, when someone tries to use such an idea in a real project, she finds out it is much harder to implement such a system than to create a diagram. If we try to use an alternative language for a particular (small) subtask in a large project, we get into a trouble with integration of language tools that have so totally different nature (for instance, strictly typed imperative language as the primary language and a typeless functional language as the secondary language). There are problems in calling conventions, in sharing global data, in using heaps etc.

Furthermore, even the fact of using two or more different programming systems within one project makes the project harder to manage. If one of the programmers does not know one of the used programming systems, she could

get into a trouble trying to build the project, to fix someone else's code etc. The difficulty of managing a project which uses two or more programming systems is so serious that this reason alone is able to prevent senior developers from making decisions of using different languages.

3 Different ways towards multiparadigm environment creation

It is obvious that an ability of using different paradigms together is attractive. There are certain difficulties though that prevent programmers from trying multiparadigm programming.

Before we introduce the new idea which hopefully allows to avoid most of the troubles, let us discuss some possible (and well-known) ways of creation of a multiparadigm environment.

We'll try to understand why each of them didn't become as widely-used as it is necessary to satisfy the need in multiparadigm programming. This will allow us to specify what do we actually want from the new technique.

3.1 Creation of another programming language

There were many attempts to create a new programming language for the purpose of multiparadigm programming (e.g., Leda[3], Oz[11] etc.). There is an unexpected trouble however. It is expensive to develop a new language. It is yet more expensive to bring the newly-developed language to the level of an industrial product so that it can be used in real software engineering practice, because this requires to support the language with useful software tools (compilers, debuggers etc.), as well as to create sufficient amount of documentation, tutorials and write and publish lots of books. But the real trouble is then to wait and see that the software engineering community doesn't tend to use the language in spite of all its advantages.

It is somewhat magic[1] when the community turns towards a new language, and this is a very rare kind of event, probably because the community is too conservative[2]. In fact it has to be. Really, starting to use a new technology requires to reeducate the personnel and change habitual methods of working. Both are very expensive, while outcome of changing languages is not so clear, specially to managers who make decisions.

This is why we decide not to try to create another language, even by extending an existing one. Enough of them are created already but that doesn't help.

[1] E.g., community preferred to use C++ with all its drawbacks while a similar but better-looking and carefully developed language Ada is in fact forgotten

[2] There's nothing bad though in this conservatism. Everything happens too fast in Computer Science so if the industry wasn't so conservative, we'd have nothing actually done.

3.2 A package of differently implemented programs

There are as well a few approaches to solve the problems of different programming languages integration within a single project. The simplest idea is just to write several programs, each in its own language, and make them interoperate using operating system's capabilities.

This kind of solution avoids problems with calling and data conventions, linking incompatibilities etc. It doesn't help tough with problems of using different programming systems within a single project.

Besides that, interoperation organization produces its own problems depending on a particular technology.

Using Unix style[3], when every program of the package has standard streams of input and output and the interoperation is done with command languages, we have to convert all the data somehow into a text representation, and then analyse the representation in another program of the package. Another drawback is that it is not always convenient to use the standard input/output streams for interoperation with another part of the package.

There are some attempts to make another, more convenient for a programmer standard way to organize interoperation of different programs, such as CORBA and COM. However, such technologies theyselves are complicated enough so that making a program to support them could be comparable in difficulty with solving the task the program is actually written for, and they are best handled with object-orientred languages, producing troubles with functional or logic languages.

3.3 Embedded interpreters

Another solution is to build an interpreter of a secondary language into the primary language. In the simplest case the interpreter is implemented as a module, which has an appropriate interface. The main program (e.g., written in C++) feeds the interpreter with the text of a module created in the secondary language (e.g., Lisp), then passes the initial data, runs the interpreter and reads the results back.

One of more advanced techniques is known as 'embeddance' of one language into another. In this case the primary language is enlarged with some certain constructs which allow to insert constructs of a secondary language into a code in the primary language. In this case we need a preprocessor which handles embeddance constructs and produces a plain code in the primary language which is then compiled in a regular way. In fact such a preprocessor replaces a foreign construct with some code which converts all necessary variables into a text, creates an appropriate query and then passes it to an interpreter, and then converts the received result as necessary. This technique is successfully applied to the case of writing database operation software using SQL-based database management system.

Such a solution, though, has many disadvantages:

[3] Unix style is mentioned as a multiparadigm environment in, e.g., [14]

– The solution does not allow us to call primary language functions from within the interpreted code. Only primary language functions can call the secondary language code, but not vice versa.
– The secondary language code is fully interpreted. That is, when the program runs, every call to the interpreted language is given in its text representation. The embedded interpreter has to perform lexical and syntactic analyses "on-the-fly" which may lead to efficiency losses.
– "The last but not least" – the results of the interpreted code calling are also presented in text form, so we need to analyse it in the main program. Just remember that analysing of strings is one of the tasks we want to avoid in C++ code and perform with an artificial intelligence language, preferably Refal, and you'll realize that something is wrong.

Besides that, mixing up interpreted and compiled execution within one program doesn't look like a fair solution anyway. It is clear that we can't avoid interpretation completely for such languages as Lisp or Prolog, but at least we could expect there will be no lexical and syntactic analyses at runtime.

3.4 Extendable interpreters

The opposite solution is to choose an interpreted language as the primary one and provide mechanisms to extend the interpreter with functions implemented in another language (usually the language in which the interpreter is initially implemented). One of the well-known examples is Tcl. Its interpreter allows to call C code which is compiled into a shared library following certain simple calling conventions.

The main drawback of this method is that the primary language **must** be interpreted which may be inappropriate in some cases.

3.5 Cross-language linkage

Having certain amount of patience, it is possible to compile and link modules written in different languages together. As it was mentioned before, this produces numerous problems with differences in calling conventions, data representation conventions etc. Furthermore, it almost always requires to make changes to the existing programming systems (for example, to reimplement compilers so that they could produce compatible object modules). As of practice, all these hardships are sufficient to prevent programmers from trying multiparadigm programming. And, anyway, we don't reach real integration of languages this way because the languages theyselves are not designed for multilanguage environment (e.g., there's a problem how to call a C function from Lisp code – how to *specify* such a call using Lisp syntax).

3.6 Compilation from one language into another

The difficulties of linking together modules implemented in two defferent languages can be reduced if one of the languages is first compiled into the other.

Many of Scheme translators actually produce C code which can then be compiled in usual manner. Thus there's no problem to link such a code with some modules implemented in C and/or C++.

We still don't know how to specify a C function call in Scheme syntax, that is, only Scheme functions can be called from C, but not vice-versa. Also, the programmer needs to understand the internal data structures of the particular implementation of Scheme in order to compose a call to a Scheme function and/or analyse the results. This is inappropriate because internal data structures are usually not well-documented.

3.7 Paradigms without a language

There's also a chance to brainstorm why do we actually want to use another language for a particular subtask, that is, what features does it have the primary language doesn't provide, and then just implement them (e.g., as a library).

Consider we use C++ as the primary language and we for some reason we feel it useful to have heterogenous lists[4] as we do in Lisp. It is possible to implement them with C++ template classes. First, we create a base class which implements the common behaviour of all items of such a list (e.g., a pointer to the next item, a pure virtual function which returns the size of this object etc.) Having this class, we define a template child of it. The template gets the type of the stored value as its parameter. Each item of such a list would be an instance of the template. Using C++ runtime type identification (RTTI) we can tell one type of an item from another when the list is handled.

Practice shows however this doesn't *completely* satisfy the programmers' needs. Each language grows a special environment where new techniques and methods appear, and these methods usually base on more than one paradigm (such as heterogenous lists). If we remember Lisp working on a C++ project, we probably won't stop with Lisp lists alone. Once we implemented the lists, the next thing we might need is Lisp mapping functions, or Lisp destructive list changing and garbage collection, and so on.

3.8 Summary

Now let's summarize what conditions do we want to meet with a new solution. We need a framework for multiparadigm programming which

1. allows to use one of the languages widely accepted by the industry as it is, i.e. with no changes to the existing compilers and other tools;
2. delivers additional paradigms as they exist in the choosen alternative language, all (or almost all) together;
3. doesn't place any limitations over interparadigm function calls and sharing data between different code;
4. doesn't require lexical and syntactic analyses of any parts of the code at runtime.

[4] Each element of the list may have its own type

In the next chapter we introduce a technique which complies the above requirements. It is discussed assuming C++ is the primary language and Lisp is the language we need to import the set of paradigms from.

4 The key idea

In order to explain the idea of a new technique, let's discuss what do we actually need from the secondary language (e.g., Lisp). Do we, for example, need its syntax? Perhaps we don't. Generally speaking, **we need the paradigms developed around the language, not the language itself**.

Lisp language implements a kind of algebra on so called S-expressions. Both program and data are built of S-expressions. The basic operations on S-expressions are:

- composition (allows to make a list or an arbitrary binary tree of S-expressions)
- decomposition (retriving elements of a list or a tree)
- evaluation (allows to treat an S-expression as a code and perform the according operations)
- lambda (allows to build an S-expression of a functional type, so-called closure, with a given list of formal parameters and a list representing the function's body)

Special type of S-expression called *symbol* (in the terminology tradition to Common Lisp [15]) or *identifier* (in the Scheme's terminology [9]) has additional operations - assigning a value, binding a value and assigning a function. In real dialects of Lisp this set is wider, but we'll limit to these 3 operations.

Besides that, there are additional basic operations (that is, operations which require one to know the internal representation of S-expressions in order to implement such an operation). Some of them are necessary to make the algebra useful (e.g. arythmetic operations on numberic S-expressions), while others are intended for the user's convenience.

The mentioned operations create an algebra on the space of S-expressions. We will denote the introduced algebra as *S-algebra*.

It is clear that S-algebra being implemented in any particular way will give us the full set of Lisp paradigms. S-algebra may be implemented without an actual Lisp interpreter. All we need is to keep it useful, that is, provide a convenient instrumental basis to operate S-expressions and apply all the necessary operations.

In some languages (including C++ and Ada) it is possible to overload standard operations such as `+`, `-`, `/` etc. This allows to implement an arbitrary algebra using very natural and convenient syntax. For example, one can implement a mathematical notion of a vector using `+` for vector addition, `-` for vector difference operation, `*` for the scalar multiplication and (for instance) `^` for vector multiplication.

In the same manner we can implement the notion of S-expression with a class (or, more precisely, with a polymorphic hierarchy of classes) and invent a certain set of operations so as to implement the whole S-algebra presented in Lisp.

5 Representation of S-algebra with C++

In this section the architecture and design of a C++ class library named InteLib [1] is explained as an illustration of the proposed idea.

5.1 S-expressions of various types

The notion of S-expression is represented with an abstract class which for historical reasons is called LTerm. In Lisp, there are S-expressions of different types (numberic constants, string constants, symbols, dotted pairs, functional objects/closures etc.) A polymorphic inheritance technique is used to represent different types of S-expressions. The LTerm hierarchy is shown at fig. 3.

In the discussed version of the library the `LTerm` class' children serve to represent various types of S-expressions:

- `LTermInteger` and `LTermFloat` represent numberic constants;
- `LTermString` represents string constants;
- `LTermLabel` is introduced to represent S-expressions whose role in the system is determined by the particular instance of the object (such as Common Lisp symbols, as well as `#t` and `#f` in Scheme);
- `LTermSymbol` represents Lisp symbols;
- `LDotPair` represents dotted pairs which Lisp lists are built of;
- `LForm` represents generic functional S-expression such as library function, user-defined function or lexical closure, Lisp macro etc.
- several additional classes to represent miscellaneous features such as hash table, i/o stream etc.

InteLib supports two types of numberic constants, namely integers and floats. Compile-time options of the library allows to choose what numberic types we actually need. It is possible to cause `LTermInteger` to use `short`, `int`, `long` or `long long` type to store the actual value, as well as `LTermFloat` can be tuned to use `float`, `double` or `long double` form of a floating point number.

Strings are implemented assuming a string constant itself is never changed. In contrast with Common Lisp, there is no vector type of S-expression in the discussed model so a string is considered as just an atomic value.

There is no special type of S-expression for single characters. They are represented with an `LTermString` object as a string which has length of 1.

Lisp symbols are implemented with class `LTermSymbol`. The class is capable to hold a reference to an object which represents the current dynamic value of the symbol and to another object which represents the function associated with the symbol. Stuff related to lexical bindings of a symbol is implemented outside

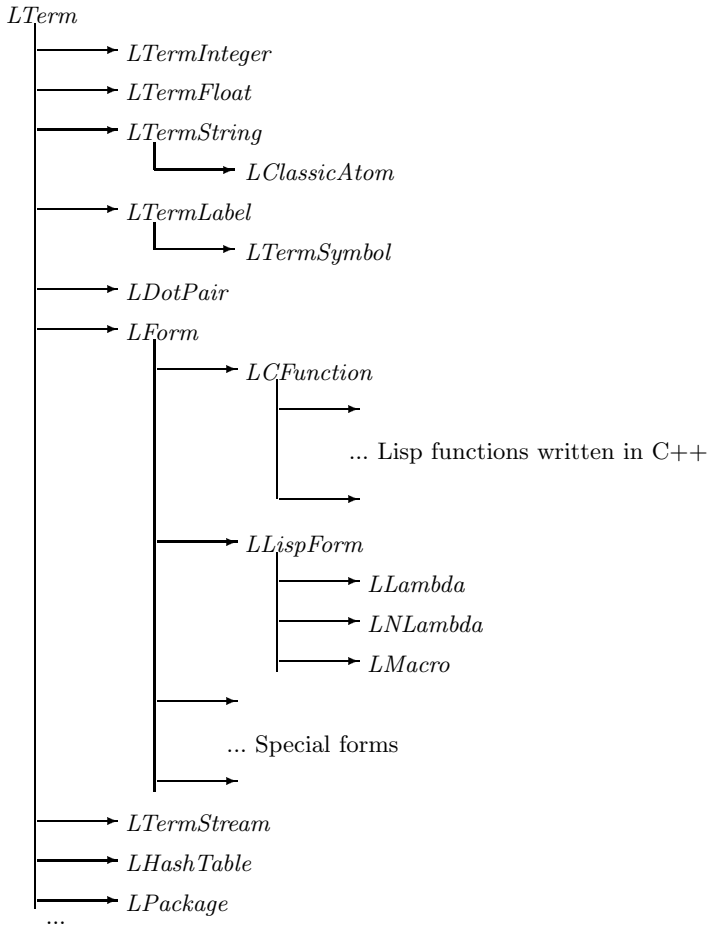

Fig. 3. LTerm classes hierarchy intended to represent S-expressions of various types

the class but is supported with its methods (those related to setting and getting the value).

The notion of an empty list may be implemented by any object of S-expression; the only condition is that the address of the object is known at the compile time because the end-of-list check is performed just by comparing pointers. Usually an object of the class `LTermSymbol` (for dialects close to Common Lisp) or LTermLabel (for Scheme-like dialects) are used for this purpose.

To represent functionals as data, `LForm` subhierarchy has been developed within the `LTerm` hierarchy. The subhierarchy has `LCFunction` and `LLispForm` classes derived directly from `LForm`. Lisp special forms are also represented with classes descended directly from `LForm`.

The `LCFunction` class represents functions implemented in C++, including all "built-in" functions such as `CAR`, `CDR`, `CONS` etc. To add a new Lisp function to the library, a programmer needs to declare a child of `LCFunction` with only one method (`DoCall`) overriden in order to implement the necessary functionality.

The `LLispForm` class is intended to represent forms defined with Lisp constructions (lambda functions, nlambda functions and macros). It has references to the lambda-list (the list of formal parameters), the function body and the lexical context[5] of the form. `LLispForm` class has child classes corresponding to different kinds of forms:

- `LLambda` (ordinary Lisp function which evaluates all its arguments and then evaluates the body in its own context);
- `LMacro` (Lisp macro evaluated as in Common Lisp);
- `LNLambda` (Lisp function which does not evaluate its arguments).

5.2 Garbage collection

Some of `LTerm`'s children are large so that it is inefficient to pass them by value. However, sometimes these objects are constructed within a function which may be called for the value as well as for its side effect so that it is no good to return the created object by pointer (if the function is called for the side effect then the constructed object goes to garbage).

In order to provide garbage collection, another class (called `LReference`) is provided. It has precisely the same size as a pointer do so that it is not so bad to pass it by value. An object of `LReference` class acts just like a pointer to an `LTerm` object having necessary operations including `*` and `->`. `LReference` is intended to be the primary interface to the library. It has a lot of constructors which allow to construct an S-expression from a value of any base C++ data type (integers, floats, strings etc).

In most cases, the objects of `LTerm` class hierarchy reside in dynamic memory and are not operated directly (altough it is possible). `LReference` objects are used to handle `LTerms`.

Besides other features, `LReference` notifies the pointed object when another pointer to it is created or an existing pointer is no longer pointing to it (e.g., it is assigned with another value or destructed). `LTerm` class performs simple reference counting and deletes the object once it has zero references.

Reference counting is choosen for its simplicity. It has well-known problems (including the problem of cyclic constructions). If it is inappropriate for a particular application to use reference counting, then any of existing C++ garbage collection libraries can be used instead. The library has a compile-time option to switch off the reference counting code.

5.3 Lexical bindings

There are also additional classes that represent (in traditional Lisp terminology) a notion of lexical context. Objects of these classes are not operated by user in

[5] The notion of lexical context is implemented with a separate class LLexicalContext

Table 1. Examples of Lisp expressions representation with C++ constructs

C++ constructs	Lisp equivalent
`(L| 25, 36, 49)`	`(25 36 49)`
`(L| "I am the walrus", 1965)`	`("I am the walrus" 1965)`
`(L| 1, 2, (L| 3, 4), 5, 6)`	`(1 2 (3 4) 5 6)`
`(L| (L| 1, 2), 3, 4)`	`((1 2) 3 4)`
`(L| MEMBER, 1, ~(L| 1, 3, 5))`	`(member 1 '(1 3 5))`
`(L| APPEND, ~(L| 10, 20), ~(L| 30, 40))`	`(append '(10 20) '(30 40))`
`(L| 1 || 2)`	`(1 . 2)`
`((L| 1, 2, 3)|| 4)`	`(1 2 3 . 4)`

most cases. The library does not, however, hide them from the user because in some cases it might be useful to create a context manually. There's always one active lexical context (possibly special null context). In order to use a context it must be activated. Then it is affected by operation of binding a value to a symbol. The context itself affects operations of assignment and retrieving a value of a symbol.

5.4 List composition operations

In Lisp there's an operation of constructing a list of an arbitrary length denoted by parentheses. The operation has variable 'arity'. For example, a construct `(1 2 3)` has 3 arguments and builds a list of 3 items - 1, 2 and 3. Besides that, there's a binary operation which builds a dotted pair, such as `(1 . 2)`. The construct `(1 2 3)` has the same effect as a superposition of 3 dotted pair constructors, like this:

```
(1 2 3)   ==   (1 . (2 . (3 . NIL)))
```

We can implement an operation similar to the Lisp's `( . )` and it will allow us to build any list of S-expressions. It is though a bit inconvenient to create lists using this operation only (imagine you couldn't use plain lists in Lisp, only dotted pairs).

Another problem is that one might want to use just a C++ constant or expression without explicit cast of it to an S-expression, e.g. '3', not a construct like `LReference(3)`, so in case of an overloaded standard operation we need at least one of operands already of LReference type, which allow a C++ compiler to understand we mean the overloaded operation, not a standard one. This requirement also prevents us from using C++ functions with unspecified number of arguments because there's no way to determine at run time what types of actual parameters do we have.

The problem is solved replacing the Lisp '( )' operation of an arbitrary list composition with two operations. First of them creates a list of one element, while second adds an element to a given list. It is clear the two operations allow to create an arbitrary list, that is, their combination has the same functionality as the Lisp '( )' operation.

The first operation always has exactly one argument, but we can't use a symbol of any standard unary operation for it because we want it to be applicable to an expression of any standard type. We also don't want to use a plain function for this purpose because parentheses would make our constructs less clear. The problem is solved with a class `LListConstructor`, which is created to be a label for a binary operation to show the compiler to apply an overloaded one instead of the built-in operation. Usually there's only one instance of `LListConstructor` named `L`. For example, an operator `L|3` returns an `LReference` object that represents Lisp construct `(3)`.

For appending a new item to a list we can overload any overloadable binary operator. For a better clarity we decide to use C++ comma (`,`) for this purpose. Left-hand operand of a comma is always an LReference representing a list. Comma destructively changes the list replacing the final `NIL` with a dotted pair of `(X . NIL)` where `X` is its right-hand operand casted to an `LReference`. This makes it possible to represent Lisp lists in C++ as shown in table 1.

There's a supplementary unary operation in most of Lisp dialects which allows to construct a list of two elements, first of which is a symbol `QUOTE` while the second is the operation's operand. The operation is usually denoted by a single quote symbol (`'`). InteLib overloads the operator `~` (tilde) for this purpose. See table 1 for an example[6].

For composing dotted pairs and dotted lists InteLib offers an operation `||`. The left-side operand must be a list (possibly of one element). The operand appearing at the right side of the operator is converted to LReference and then the operation replaces the last `NIL` of the given list with whatever it constructed from the right-side operand. See table 1 for an example.

Note the parentheses in the last example. They appear because the comma operator has lower precedence than `||`.

5.5 Operations implemented as regular methods

The most important operation on S-expressions is the *evaluation* of an S-expression. Evaluation is an unary function which maps from $S_x \backslash S_{ux}$ to S_x where S_x is a space of all possible S-expressions and S_{ux} is a set of 'unevaluable' S-expressions (such as closures). Performing evaluation of an S-expression one can also get a side effect.

All constants evaluate to themselves with no side effects. Variables evaluate to their values, if any. Evaluation of an unbound variable generates an error.

Evaluation of a list interpretes the first element as an instruction what functional object to apply to the rest of the list as a list of parameters. In most cases, the resting elements of the list are evaluated and the appropriate function is applied to a list built of the results. The first element of the list must be either a symbol which has an associated functional object or a Lambda-list.

[6] The operator is oveloaded for LReference class so it is possible to apply it to a list or a Lisp symbol, but one can't use it with strings or numberic constants. It is unnecessary anyway to quote them since they always evaluate to themselves

It looks like we need to implement two operations in order to support the evaluation: the evaluation itself (as a polymorphic method of the `LTerm` class) and an operation of *application* of a functional S-expression (a one that belongs to `LForm` subhierarchy) to a list of parameters. The two operations are implemented as methods called '`Evaluate`' and '`Call`', respectively. The '`Call`' method generates an error when called for an object of a non-functional type. '`Evaluate`' generates an error when it is impossible to evaluate the given S-expression. The `IsSelfEvaluated` method allows to determine whether the object represents a constant that always evaluates to itself.

Besides that, there are methods '`Car`' and '`Cdr`' in LTerm class. They return the respective cells of a dotted pair when called for an `LDotPair` object. For an object that represents empty list both methods return empty list. Calling these methods for a non-list object will cause an error.

Another important method named `TextRepresentation` allows to create a human-readable representation of any given S-expression.

It is well known that there are 3 different predicates of equality in Lisp, called `EQ`, `EQL` and `EQUAL`. `EQ` predicate is the simplest one, it just compares two addresses. `EQL` is a bit more flexible. Two objects may be not the same while representing the same value (e.g. two instances of an integer constant 2). In order to make it possible to implement `EQL` predicate for any `LTerm` object there's a virtual method `SpecificEql` which returns false by default. Implementation of `EQL` checks for equality of addresses first, and only if the objects are not `EQ`, it calls `SpecificEql` for one of them passing the other as an argument, so that it doesn't cause a misbehaviour when `SpecificEql` returns false when the compating objects are the same, that is, equal in the sence of `EQ`.

Another important operation, Lambda, is implemented by a constructor of `LLambda` class. For example, expression

```
LReference(new LLambda(NULL, (L| A),
                       (L| (L| PLUS, A, 1))))
```

creates a closure with `NULL` lexical context. The closure takes a numberic S-expression and returns a number which is greater by one. In Lisp such a closure would be represented as `(lambda (a) (plus a 1))`. In order to create a real closure that has lexically bound variables, the appropriate lexical context must be passes as the first argument of the `LLambda` constructor.

There are other methods in classes of the library which a provided mostly for user's convenience. They include typecasting operations, which allow to convert a constant S-expression into a base C++ value. For example,

```
LReference(3)->GetInteger();
```

will return 3, while

```
LReference("Hello world")->GetString();
```

will return a pointer to a constant string `"Hello world"`. Calling such a method for a wrong type of S-expression causes an error.

5.6 Operations performed by standard Lisp functions

Standard, or built-in, functions play a key role in Lisp functionality providing a basis for building programs. They can also be thought as operations on S-expressions, that is, as elements of S-algebra.

In Lisp there are symbols that initially have associated built-in functions. InteLib doesn't provide such symbols in order to allow a user to use whatever names she wants for these symbols. The functional objects representing well-known Lisp functions are direct children of `LCFunction` class (for functions that evaluate all arguments) or of `LForm` class (for special forms). They usually have names such as `LFunctionCar`, `LFunctionCons`, `LFunctionLet`, `LFunctionDefun` and so on.

For convenience there is a generic class

```
template<class F> class LFunctionalSymbol
```

whose argument must be a class that represents a particular function. An instance of `LFunctionalSymbol` differs from `LSymbol` in that its constructor creates an object of the given finctional class and lets it be the associated function of the symbol. For example, one might want to add the following declaration to the program:

```
LFunctionalSymbol<LFunctionCar> CAR("CAR");
LFunctionalSymbol<LFunctionCdr> CDR("CDR");
LFunctionalSymbol<LFunctionCons> CONS("CONS");
LFunctionalSymbol<LFunctionCond> COND("COND");
LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
```

etc. As usual, the constructor's argument sets the textual name of the symbol which is used by `TextRepresentation` method.

Consider, for example, the following Lisp code:

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1)
                           (car tree2))
               (isomorphic (cdr tree1)
                           (cdr tree2))))))
```

One can write the module shown at fig. 4 to do the same thing in C++. The module compiles with an ordinary C++ compiler without any additional preprocessing. The symbol `ISOMORPHIC` is public and can therefore be used in other modules.

Please note there are no definitions of symbols `T` and `NIL`. They are provided by the library as well as symbols `QUOTE` and `LAMBDA`. The library needs symbols `T`, `NIL` and `QUOTE` because it is possible to obtain them from certain operations without mentioning them in a program. Consider the following example:

```
//       File isomorph.cpp
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
  static LSymbol TREE1("TREE1");
  static LSymbol TREE2("TREE2");
  static LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
  static LFunctionalSymbol<LFunctionCond> COND("COND");
  static LFunctionalSymbol<LFunctionAtom> ATOM("ATOM");
  static LFunctionalSymbol<LFunctionAnd> AND("AND");
  static LFunctionalSymbol<LFunctionCar> CAR("CAR");
  static LFunctionalSymbol<LFunctionCdr> CDR("CDR");
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND,
      (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
        (L|(L|ATOM, TREE2), NIL),
        (L|T, (L|AND,
          (L|ISOMORPHIC, (L|CAR, TREE1), (L|CAR, TREE2)),
          (L|ISOMORPHIC, (L|CDR, TREE1), (L|CDR, TREE2)))))).Evaluate();
}
//      end of file
```

Fig. 4. Example of a C++ module that defines a function in a manner of Lisp

```
(eql 1 2)   ->  nil
(eql 1 1)   ->  t
(car ''a)   ->  quote
```

The symbol `LAMBDA` can't be obtained in such a way, but it has special meaning regardless of it's possible value and/or associated function, so we have to rely on the symbol itself (that is, for example, on the object's address). That's why these 4 symbols are provided by the library in contrast with all the other well-known symbols.

It is important to understand that there's no Lisp as such in the C++ module shown at the fig. 4. The module is written in C++ language. The compiler knows nothing about special meaning of all these commas and vertical bars; they are handled as functions just like in any program which overloads standard operators. Thus we can say we made no changes to the primary language, at the same time allowing a programmer to use paradigms from another (secondary) language. Only paradigms are imported from Lisp, not the language itself. In other words, we **import S-algebra which brings us the paradigms of Lisp without Lisp language as such.**

6 Translation from Lisp to C++

The primary goal of the InteLib library is to bring Lisp paradigms to C++. However, as a side effect it opens a clear way to translation of Lisp code into

C++. The translator is made which uses simple rules of transformation of the code. Besides that, the translator generates the necessary definitions of symbols and performs some other tasks as to allow using several Lisp modules within a single project. Since the translator is not the main goal of the project, we don't explain it in details in this paper; we only give a short description.

The translator takes one or more Lisp files and proudces a C++ module (that is, a "source" file and a header file). The translator understands a certain set of so-called translation directives (top level forms beginning with a token `%%%`), which allow to control how names are translated etc.

The character set for C++ identifiers differs from the traditional one for Lisp symbols. C++ identifiers are case-sensitive and can consist of letters, digits and the underline symbol. The first character of identifier must be a letter or the underline, not a digit. Lisp symbols are case insensitive and may be built of letters, digits and various symols such as `+`, `-`, `*`, `_`, `%` etc. so we need a certain translation algorythm.

The fact that Lisp symbols are case sensitive and C++ identifiers are not is very helpful in translation of names from Lisp to C++. This allows us to bring all letters in a Lisp symbol to the upper case and leave the lower-case letters to represent all these pluses, dashes and other symbols that are not allowed in C++ identifiers. For instance, the Lisp symbol `read-char` might be represented as `READdashCHAR`. If a symbol's name begins from a digit (which is illegal in C++), we prepend it with a lowercase letter, for instance - "`d`". The symbol `7seas` having been translated this way becomes the identifier `d7SEAS`.

There are some exceptions from the general rules. For example, the Lisp symbol `null` would be translated to `NULL` producing a name conflict with some of the standard header files. That's why it is translated as `lNULL` (as specified by an appropriate translation directive). User can add her own exceptions, rules etc.

As to experience, using of the translator is good when a whole module is inplemented in Lisp because the traditional Lisp syntax is more convenient than the introduced C++ implementation of S-algebra. It is still possible, however, to avoid using the translator.

The dialect of Lisp recognized by the translator was named InteLib Lisp. It is a very short and simple dialect designed keeping in mind that it is to be used as a secondary language. It ommits many features of modern Lisp dialects because they are considered not to be essential.

7 Logic programming

The explained technique can be applied to secondary languages other than Lisp. One of the obvious ways of further development is to apply it to one of the logic programming languages.

As of now, an attempt is done to apply the technique to a very restrictive subset of Prolog[4]. The Prolog part of the library, unlike the Lisp part, has

primarily a demonstration value; creating a library useful in real programming practice is the subject of further work.

The implemented subset of Prolog has no dynamic data structures (lists and functors), just like in Datalog[5]. Unlike Datalog, the implemented dialect has the *cut operation*.

Prolog machines operate on data which is similar to S-expressions (in fact, the only difference is functors). Creating a model of a Prolog machine, a decision was made to reuse the classes already implemented to represent Lisp data[7].

To represent the notion of *predicate*, `DlAbstractPredicate` class is invented. It has a pure abstract method named `DlCreateAbstractIterator` which is intended to create an iterator to fetch, one by one, solutions of a given predicate provided with the appropriate number of arguments.

To create atoms of given predicate (e.g., having predicate `father`, create the atom `father(john, X)`), the class provides `operator()` for 0, 1, 2, ..., 10 arguments of the type LReference[8].

Another class, derived from `DlAbstractPredicate` and named `DlPredicate`, represents the predicate which is the part of Prolog program (that is, a predicate formed of clauses of goals).

Prolog atoms (constructs such as `father(john, mary)`, `vertex(X)`, etc.) are represented with the `DlAtom` class which is in fact just a pair of a predicate and an argument list. Atoms are usually created within functions, locally, so another smart pointer is necessary. It is named `DlAtomRef`. This smart pointer is used as the primary interface to the `DlAtom` class. One of the most important operations of `DlAtomRef` is `operator<<=()` which is used instead of the well-known Prolog symbol `:-`. The operator adds another clause to the appropriate predicate and returns a reference to the object of the class `DlPredicate::DlClause`. That object, in turn, has `operator,()` to add goals (atoms) to it.

Prolog variables are represented using class `DlVariable`.

Using all these classes and operations, we can represent the Prolog clause

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

with C++ expression

```
grandfather(X, Y) <<= father(X, Z), father(Z, Y);
```

where `X`, `Y` and `Z` are objects of the class `DlVariable`. Facts such as

```
father(john, george).
```

are represented using another form of the operator `<<=`:

```
father(john, george) <<= true;
```

[7] So the dialect in fact can handle lists, but it still treats them as atomic data values, e.g., when doing unification

[8] There is no operator with variable parameters list, because LReference objects, being objects of a class, can't be passed through ... in C++.

Prolog code example:

```
father(john, george).
father(george, alex).
father(alex, alan).
father(alan, poul).
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Equal C++ code:

```
#include "il_dlog.h"
LSymbol john("john"), george("george"),
  alex("alex"), alan("alan"), poul("poul");
DlVariable X("X"), Y("Y"), Z("Z");
DlPredicate father, grandfather;
void PrologInit_father_grandfather() {
    father(john, george) <<= true;
    father(george, alex) <<= true;
    father(alex, alan) <<= true;
    father(alan, poul) <<= true;
    grandfather(X, Y) <<= father(X, Z), father(Z, Y);
}
```

Fig. 5. Prolog-like C++ code example

Fig. 5 shows an example of a Prolog-like C++ code which uses the predicates `father` and `grandfather`. Now, the code

```
iter = grandfather(X, Y).CreateIterator();
bool rc;
do {
  PDlSubstitution solution;
  rc = iter->NextSolution(solution);
  if (rc) {
    printf("%s\n", solution->TextRepresentation().c_str());
  }
} while (rc);
```

will print the following solutions list:

```
{X/john Y/alex}
{X/george Y/alan}
{X/alex Y/poul}
```

As we already noted before, the implemented model doesn't do unification of dynamic data structures though Lisp lists can be operated with, considering them atomic datums. To produce a more interesting demo, let's add a built-in predicate named `DLCONS(car, cdr, cons)`. The predicate is implemented by another class derived from `DlAbstractPredicate`, named `DlPredicateCons`.

The predicate is able to work having any of its arguments specified or unspecified (that is, a variable is given instead of a value).

Another useful predicate is `DlPredicateLispcall` (to call the Lisp machine explained before).

The object `DlCut` is a special value of `DlAtomRef` which represents the cut operator.

Note also that a `DlPredicate` without any clauses always fails, so to implement an *always failing* goal we can just create an empty `DlPredicate`.

Having all these objects, we are ready to write a simple program which finds a path in a given graph (fig. 6).

Now, if we create the appropriate iterator with

```
iter = Shortpath(2, 4, X, 2).CreateIterator();
```

the solution finding code like the one shown above will print the solution

```
{X/(2 1 4)}
```

Unlike the Lisp part of InteLib which is already useful in some practical cases, the explained Prolog part is only a simple demo. It is planned to implement a more practically useful library in the close future.

8 Conclusions

The most important advantage of the proposed technique is that there's no need for two programming systems within a project. The existing C++ compiler is always used, and the only thing required to use the technique is a C++ class library which has a relatively simple imterface.

It is also possible to use another primary language. The only requirement to it is the possibility of overloading of standard operations. In particular, Ada95 may be used as the primary language as well (at least for modelling Lisp as the secondary language). Implementation of the appropriate library for Ada95 might be one of the further work goals.

References

[1] E. Bolshakova and A. Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.

[2] G. Booch. *Object-oriented Analyses and Design*. Addison-Wesley, Reading, Massachusets, second edition, 1994.

[3] T. A. Budd. *Multy-Paradigm Programming in LEDA*. Addison-Wesley, Reading, Massachusets, 1995.

[4] A. Calmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983.

```
LListConstructor L;
DlPredicateCons DLCONS;
DlPredicateLispcall DLLISPCALL;
DlAbstractPredicateIterator *iter;
DlVariable X("X");
DlVariable Y("Y");
DlVariable Z("Z");
DlVariable P("P");
DlVariable N("N");
DlVariable V1("V1");
DlVariable V2("V2");
DlVariable V3("V3");
DlPredicate Edge("Edge");
DlPredicate Edge2("Edge2");
DlPredicate Member("Member");
DlPredicate Shortpath("Shortpath");
DlPredicate Fail("Fail");
Member(X, Y) <<= DLCONS(X, V2, Y);
Member(X, Y) <<= DLCONS(V1, V2, Y), Member(X, V2);
Edge(1, 2) <<= true;
Edge(1, 3) <<= true;
Edge(1, 4) <<= true;
Edge(1, 5) <<= true;
Edge(5, 3) <<= true;
Edge2(X, Y) <<= Edge(X, Y);
Edge2(X, Y) <<= Edge(Y, X);
Shortpath(X, Y, P, 1) <<=
             DlCut,
             Edge2(X, Y),
             DLCONS(Y, L, V1),
             DLCONS(X, V1, P);
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|lt, N, 1), T),
             DlCut,
             Fail();
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|minus, N, 1), V1),
             Shortpath(Z, Y, V2, V1),
             Edge2(X, Z),
             DLCONS(X, V2, P);
```

Fig. 6. Graph path finding program

[5] S. Ceri, G. Gottlob, and L. Tanka. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[6] U. W. Eisenecker. Future trends in multi-paradigm programming. Position Paper for the ECOOP'98 Panel on Multi-Paradigm Programming, 1998.

[7] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Reading, Massachusets, 1998.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusets, 1983.

[9] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on Algorithmic Language Scheme, 1998.

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[11] M. Müller, T. Müller, and P. Van Roy. Multiparadigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Workshop on the Future of Logic Programming*. International Logic Programming Symposium, 1995.

[12] J. Robinson. Logic programming - past, present and future. *New Generation Computing*, 1:107–121, 1983.

[13] D. Spinellis, S. Drossoupoulou, and S. Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In J. Gutknecht, editor, *Programming Languages and System Architectures International Conference*, volume 782 of *Lecture Notes in Computer Science*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag.

[14] D. D. Spinellis. *Programming paradigms as object classes: a structuring mechanism for multiparadigm programming*. PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

[15] G. L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusets, third edition, 1997.

[17] V. Turchin. *REFAL-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989.

[18] D. A. Turner. Miranda – a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings of the Conference of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, 1985. Springer-Verlag.

Program Templates:
Expression Templates Applied to Program Evaluation

Francis Maes

EPITA Research and Development Laboratory,
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France,
francis.maes@lrde.epita.fr,
WWW home page: http://lrde.epita.fr/

Abstract. The C++ language provides a two-layer execution model: static execution of meta-programs and dynamic execution of resulting programs. The Expression Templates technique takes advantage of this dual execution model through the construction of C++ types expressing simple arithmetic formulas. Our intent is to extend this technique to a whole programming language. The Tiger language is a small, imperative language with types, variables, arrays, records, flow control structures and nested functions. The first step is to show how to express a Tiger program as a C++ type. The second step concerns operational analysis which is done through the use of meta-programs. Finally an implementation of our Tiger evaluator is proposed.
Our technique goes much deeper than the Expression Templates one. It shows how the generative power of C++ meta-programming can be used in order to compile abstract syntax trees of a fully featured programming language.

1 Introduction

During the compilation process, an input program expressed in textual form is transformed by successive steps into executable code. As in any language, a C++ program will basically be evaluated during its execution. The interesting particularity of C++ is its ability to do some computations at compile-time using template constructions (the so-called meta-programs, see [12], [3], [7] and appendix A for an example). This two-layer execution model corresponds to the usual concept of static (compile-time) and dynamic (execution-time) processing.

In C++, there is a technique called Expression Templates described by [11], which allows the exploitation of this two-layer execution model. This technique relies on transformations of simple arithmetic expressions at compile-time to increase the performances of the executable code. Moreover some evaluation can be done entirely statically with mechanisms such as constant propagation. This way, some computations usually done at execution-time are processed at compile-time.

The Expression Templates technique is based on the use of template classes. In order to work on expressions, we need a structural description of them. This

is done by building a type that reflects the abstract syntax tree (AST) of the expression. Each node of this tree will be translated into a template class whose arguments are the node subtrees.

Usually, a program written in any language can also be expressed as an abstract syntax tree. The next natural step is to wonder whether it is possible to extend the Expression Templates technique to a whole programming language. Expressing a full program with a C++ type reflecting its AST could thus be made possible. In the remainder of this paper, this type will be called the TAT (Tree As Type). A TAT is a representation of an AST using a C++ type formalism.

Expressing a program in the TAT formalism would allow us to adapt the Expression Templates evaluation method to a whole program and therefore to take advantage of the two-layer execution model of C++ (see [5]). The entire process of compiling and executing a program expressed as a TAT corresponds to its evaluation.

To experiment this idea, we have to choose a programming language that does not have this two-layer execution model. We want this language to be simple and to have few constructions. Nevertheless, this language must at least include types, functions, records, arrays and flow control constructions. Tiger, a language defined by [1], corresponds to our needs: with only 40 rules in its EBNF grammar, it respects all our conditions.

This work is a proof of concept. No-one had previously mapped an entire language to a C++ meta-program. Those that consider C++ expression templates for prototype implementations should be interested in this project. Moreover, the C++ metalanguage is here introduced as an intermediate language. This point of view is different from the current trend of supporting meta-programming by designing metalanguages as extensions of existing programming languages. Our work initially inspired by Expression Templates goes very deeply into the possibilities of C++ meta-programs using several techniques discovered recently.

This paper begins with an overview of related work. Next, section 3 introduces the Tiger language, followed by a description of our architecture. Our first objective is to translate Tiger programs into TATs. When trying to do this, several problems arise (e.g. expressing lists). These are developed in section 4. Our second objective is to do some static processing on this TAT. This will require a structure called environment, and a form of static pointers detailed in section 5. Finally we want to evaluate a Tiger program expressed as a TAT using the C++ two-layer execution model. The implementation which allows this is described in section 6. This is followed by some interesting results related to this new technique. This paper will finish with a discussion about the possibilities of such mechanisms.

2 Related work

Our work is based on Expression Template. The Expression Template is at the basis of our work. This technique described by [11] has many known interests.

In particular it allows to build the static AST of a C++ expression. This allows C++ meta-programs to work on C++ expressions seen as types. This can be useful for:

- **Rewriting statements into equivalent (but more efficient) ones**. This was the original intent of Expression Templates. This technique was first used to evaluate vector and matrix expressions in a single pass, without temporaries.
- **Building lambda terms**. Several libraries for doing functionnal programming in C++ are based on Expression Templates. Thanks to C++ meta-programs, several functionnal operations are possible on these lambda terms. The Fact library ([9]) provides typical functional features such as currying, lambda expressions and lazy evaluation in C++. The Boost package also includes a library specialized in lambda expressions: the Boost Lambda Library ([6]). FC++ ([8]) is a similar library inspired by the Haskell language. Our work has something to do with lambda term manipulations: we also manipulate TATs. But our intent is not to do functionnal operations on a TAT but to compile a whole program including functions and variables declarations.
- **Building any other structured expressions**, such as the [4] library which uses Expression Template in order to build EBNF rules. C++ meta-programs are then used to transform a grammar into a usable parser. In this library, C++ meta-programs deal with complex operations such as in our work.

The Expression Template is very useful but a bit complex to implement. PETE ([2]) is a tool that aims at generating the needed code. Fact is built on top of PETE. This tool could help us to build a C++ front-end to our compiler. The idea of using template constructions in compilers has already been used for building a java compiler, see [10].

3 Tiger evaluation and compilation

3.1 Tiger constructions

Tiger is an Algol-style language with a functional flavor. Two kinds of construction exist: declarations and typed expressions. Declarations are of three kinds: type, variable and function declarations. Four basic types exist: integers, strings, nil and void. New types can be built with records and arrays. Existing types can be renamed by a typedef mechanism. Tiger is not a first-order functionnal language: functions cannot be passed as parameters, neither as results.

Tiger has a nested `let-in-end` construction which makes it possible to declare nested scopes. A particular case of this is the ability to declare nested functions.

Except declarations, everything in Tiger is an expression: literals (strings and integers), unary and binary operations, left-values, function calls, array and record instantiations and flow control constructions: `if-then-else`, `while-do`, `for-to-do`, `break`.

3.2 Architecture

We use a front-end program which parses Tiger and does the semantic analysis: type checking, scopes and bindings. The output of this front-end is a C++ program which declares a TAT. Our front-end is based on techniques explained by [1].

The interesting thing is the remaining work: the program evaluation. This task is done in C++ through the static and the dynamic processing.

Our front-end associated with the C++ static processor is a compilation chain. Indeed the input of this chain is a textual Tiger program, and its output is an executable program.

3.3 Comparison with a standard compiler

A usual object oriented compiler first parses the program. It provides AST classes that are dynamically instantiated in order to build the programs abstract tree. At this point until the end of the compilation, successive transformations are applied until getting the executable code.

In our case, we provide a set of template classes corresponding to each node of the AST. During the compilation of a Tiger program, these templates are filled by our front-end giving us the TAT. At this point, the C++ compiler does successive transformations until getting the executable code.

An analogy can easily be done between our Tiger compiler and a standard compiler. Where a standard compiler provides AST classes, we provide AST meta-classes. Where a standard compiler builds an AST expressed as objects, we build an AST expressed as a type (the TAT). A standard compiler provides classes for operational analysis, we provide meta-classes to do this work.

It has been shown that a Turing machine could be constructed with template constructs ([12]). Any work traditionally done by a standard compiler can theoretically be done with C++ meta-programs. The method that we present should thus be adaptable to any other language. The only restrictions are the C++ compilation times and memory use.

4 Translation into TAT

Let us return to the Expression Templates technique with the following Tiger program:

```
( 5 * 10 + 1 )
```

Since the Expression Templates technique was originally used to describe and evaluate simple expressions (literals, variables, unary, binary and potentially n-ary operations), such examples can easily be constructed with it. Here is an example of TAT corresponding to the previous example:

Listing 4.1. A simple TAT

```
typedef BinOp< BinOp< ConstInt<5>, ConstInt<10>, Times >,
                                   ConstInt<1>, Plus >
                program_t;
```

However this covers a very small part of the whole programming language. Important features such as type declarations, function declarations and calls, or flow control cannot be expressed. Moreover, Tiger expressions are typed: we want our compiler to be able to evaluate and work on typed-expressions. When trying to translate more complex examples into TATs, different problems arise such as the list problem, or the reference problem.

4.1 The list problem

Let us consider this Tiger example:

Listing 4.2. Two functions

```
let
  function double(x : int) : int = 2 * x
  function sum(a : int, b : int, c : int): int = a + b + c
in
  double(30) - sum(6, 1, 2)
end
```

When building this program's TAT, we need to express lists: declaration lists, function formals lists, and function call arguments lists. The usual way to do this is to use recursive lists. A recursive list is defined as empty or as a head element followed by a tail list.

This can be transposed into C++ with the static list technique described by [12]. We use a template class List, which parameters are the first element (a type), and the remaining list. A class EmptyList is used to mark the end of the list. With this notation, we can express lists as types. For example, in `sum (6, 1, 2)`, the argument list can be expressed with the following TAT:

```
List< ConstInt<6>,
  List< ConstInt<1>,
    List< ConstInt<2>,
      EmptyList
        >
      >
    >
```

The full TAT conversion of a similar sample is given in the next section. Static lists, which are a particular case of trees, will be used extensively in the remaining of this paper: this is our first addition to the Expression Templates technique.

4.2 The reference problem

The following simple example illustrate the reference problem:

Listing 4.3. Two variables addition

```
let
  var i : int := 80
  var j : int := 6
in
  i + j
end
```

The expression `i` refers to the variable declaration `var i : int := 80`. The same way, the declaration `var i : int := 80` refers to the builtin type `int`. This example demonstrates that we cannot consider programs as simple trees. The main structure acts as a tree, but the implicit relations by reference transforms this tree into a DAG (direct acyclic graph).

The TAT has to describe a tree plus some graph relations between a declaration and its uses. This is the main difficulty compared to the Expression Templates technique. Without a reference mechanism, we cannot express concepts such as types or functions.

Each time a declaration is referred, we need a pointer to it. The following part shows how to solve this: each declaration will have a location in an evaluation environment.

5 Evaluation Environment

At every point in the program, there is a set of active declarations which can be used. An expression such as `i + j` (listing 4.3), or `double(30) - sum(6, 1, 2)` (listing 4.2) cannot be evaluated without the declaration context: we need to maintain an environment at evaluation time.

Tiger defines some builtin types and functions. These declarations, visible at every point in every Tiger program, will be the initial state of our environment. Declarations that have the same visibility are grouped into scopes. In the remainder of this paper, the list of declarations of the same scope is called a *chunk*.

The main operations we need on this environment are pushing and popping chunks. Moreover, we need a way to extract a declaration, given its chunk and its location in the chunk.

New declarations are introduced with the `let-in-end` structure, which is composed of two parts. A first declarative part, located between `let` and `in`, allows declaring a chunk. The second part, is an expression, in which we can use previous declarations. Evaluating the whole structure is done by pushing the chunk into environment, evaluating the expression and finally popping the chunk.

The environment can also be modified by a function call: when this occurs the evaluation point is changed. This implies that the set of active declarations changes.

Listing 4.4. A function call

```
let
  function double(x : int) : int = 2 * x
in
  let
    var i : int := 17
  in
    double(i) + i
  end
end
```

In the above example, the function call is evaluated the following way:

1. Evaluate function parameters: here i = 17.
2. Initialize formal values: x ← i
3. Pop declarations introduced between the function declaration and the function call: this restores the environment of the function implementation. In our case: pop the chunk containing `var i : int := 17`, as the function double does not know this declaration.
4. Push formals declarations. Here: push a chunk containing `x : int`.
5. Evaluate the function body: `(2 * x)`
6. Restore callers environment: `x` does not exist any more, `i` is reintroduced.

At this point, a stack seems to be appropriate for our needs. This stack will be filled with declaration chunks. A declaration chunk simply contains the corresponding part of the TAT. At a given evaluation point, each visible declaration is located with a pair of indexes: the index of the chunk, and the index of the declaration in the chunk. So a simple pair of indexes is enough to refer to a declaration.

The example 4.3 can now be translated into the following TAT:

```
LetInEnd<
  List< Var< ConstInt< 80 >, builtin_types , int_type >,
    List< Var< ConstInt< 6 >, builtin_types , int_type >,
      EmptyList > >,
  BinOp< SimpleVar< 0, 0 >, SimpleVar< 0, 1 >, Plus >
>
```

The pair $< 0, 0 >$ refers to the first declaration of the first chunk, which corresponds to `var i:= 80`. The pair $< 0, 1 >$ refers to `var j:= 6`. `builtin_types` and `int_type` are predefined integer values, which identify the builtin `int` Tiger type. This mechanism of environment and location pair is a form of static pointers.

We are also able to translate example 4.4:

```
LetInEnd<                                    let
  List< Function< List<                       function double(
      TypeLnk< builtin_types, 1 > >,                      x : int) =
    BinOp< ConstInt< 2 >,                        2
      SimpleVar<1, 0 >, Times >,                  * x
    0 > >,
   LetInEnd<                                 in
     List< Variable< ConstInt< 17 >,           let
                builtin_types, 1 > >,            var i : int := 17
                                               in
       BinOp< FuncCall< 0, 0,                    double(
           List< SimpleVar< 1, 0 > > >,                    i)
       SimpleVar< 1, 0 >, Plus >                     + i
    >                                          end
>                                            end
```

Let's remember the goal: translating an AST into a C++ type (the TAT), so that the compiler can work on this type. In the proposed implementation, the environment related computations are done at compile-time. Meta-programming techniques will allow us to reduce the execution-time work considerably.

6 Implementation

The basis of the Expression Templates technique is to write a template class per kind of node available in the AST. The parameters of this template are the node subtrees. Each of these template classes correspond to a node of the AST.

These template classes fulfill two roles: first they express the AST information. This is implicitly done with class organization into the TAT. Second, our classes must provide evaluation code.

In the case of expressions, this consists on two tasks: the type calculation, and the value calculation. The declaration classes provide some other services such as common operations for types.

Apart from AST meta-classes, we also need to provide meta-code to perform some static processing. This corresponds to the set of operations related to the evaluation environment.

6.1 Global organization

Two kinds of classes have to be written: expression classes and declaration classes. Declarations will be further distinguished via classes specialized for type, variable and function declarations. Moreover, the implementation also includes the environment mechanism, and tools for its manipulation.

Note that the base classes `AstNode`, `Expression`, `Declaration`, `TypeDec`... are only used for some static checking. These classes are not very interesting, and will not be detailed in this paper.

6.2 Expression classes

As in the Expression Templates technique, each Expression class will implement an evaluation method. These methods are inlined, so that the C++ compiler can build efficient evaluation code.

The main difference with Expression Templates is due to the evaluation environment: The evaluation method depends on the current environment. Another striking difference is that expressions are typed. Evaluating an expression consists in computing both its type and value. We want expression types to be evaluated statically: this work will be done through typedefs. All the typed values that we manipulate are represented with four bytes. In order to simplify, we decided to represent all variables with the `void*` type. This lead us to the following model adopted by all expression classes:

```
// var_t represent a non-typed value.
typedef void* var_t;

// Here comes the template parameters: the TAT subtrees.
template< ... >
struct AnExpression: public Expression
{
  // Evaluation is dependent of current environment.
  template<class T_env>
  struct eval
  {
    // statically compute the expression type
    typedef ... T;

    // inline method that evaluates the expression value
    inline var_t doit ()    { ... }
  };
};
```

```
template<signed Value>
struct ConstInt: public Expression
{
  template<class T_env>
  struct eval
  {
    typedef IntType         T;
    inline var_t doit ()    {return (var_t)Value;}
  };
};
```

ConstInt $< 123 >$ is a TAT: its value and type can be evaluated:

```
  typedef ConstInt<123>  program_t;

  var_t value = program_t::eval< initial_env_t >::doit();
  typedef program_t::eval< initial_env_t >::T  type;
```

Notice that "::" is C++ for Java "."

Two types are predefined:

- `var_t` represents all Tiger variables. For example, an `int` can directly be casted into a `var_t` (these two types have the same size: four bytes). Most of time a `var_t` corresponds to a record pointer or an array pointer.
- `initial_env_t` corresponds to the Tiger builtin environment: builtin types such as `IntType` or `StringType`, and builtin functions (`print`, `ord`, `concat`...).

The TAT given in listing 4.1 can now be evaluated. Here is the template expansion chain that leaded to the result: 51.

```
1. program_t::eval< initial_env_t >::doit()
2. BinOP< ConstInt<5>, ConstInt<10>, Times >
                ::eval< initial_env_t >::doit() +
     ConstInt<1>::eval< initial_env_t >::doit();
3. ConstInt<5>::eval< initial_env_t >::doit() *
   ConstInt<10>::eval< initial_env_t >::doit() + 1;
4. 5 * 10 + 1
5. 51
```

6.3 Declaration classes

The first role of declaration classes is to store information relative to the declaration. For example a variable declaration must store its type and initial value. This is done with template parameters exactly as above. The second role of declaration classes depends on the kind of declaration. For variables and functions, only some utility functions are implemented. The type classes do more things: their second role is to implement all operations related to the type: assignment, comparison, creation and destruction. These operations can depend on the environment. This is for example the case for an array, which refers to the type of its elements.

Here is the model of type declaration classes:

```
struct AType: public TypeDec
{
  // Type eval depends on environment
  template<class T_env>
  struct eval
  {
    // Common operations are implemented here
    void create(var_t& v);
    void destroy(var_t v);
    void assign(var_t& left, var_t right);
    int compare(var_t left, var_t right);
  };
};
```

Such classes are implemented for `VoidType`, `IntType`, `StringType`, `ArrayType` and `RecordType`.

Each new type definition in a Tiger program, will result in new type operations. Our Tiger compiler generates evaluation code, but also operations code. In order to emphasis on this contribution, we chose to implement assignment and comparison as structural. At the contrary to the Tiger specifications, when two records are compared, this is done member by member. When an array is assigned, the all content is copied.

6.4 Program Environment

We have seen that type and expression evaluations depend on an environment, through the type identified by `T_env` in the previous code samples. We want the environment to be computed statically: we need an implementation which allows to push, pop, and retrieve declarations at compile-time. Therefore we use again static lists: an environment is implemented as a static list of declaration chunks. A declaration chunk is a part the TAT which is also a static list. This construction allows us to manipulate the environment:

Pushing and popping declaration chunks is done with typedefs:

```
// Push T_new_chunk on T_env, yielding T_new_env.
typedef List< T_new_chunk, T_env >       T_new_env;

// Pop an element of T_env, yielding T_new_env.
typedef T_env::tail                      T_new_env;
```

Environment access is done with a template class and a specialization:

```
template<class T_env, unsigned N>
struct ListGet
{
  typedef ListGet<T_env::tail, N - 1>::T  T;
};

template<class T_env>
struct ListGet<T_env, 0>
{
  typedef T_env::head      T;
};

// Access to the chunk number 3.
typedef typename ListGet<T_env, 3>::T            T_chunk_3;
// Access to the declaration number 1 of this chunk.
typedef typename ListGet<T_chunk_3, 1>::T        T_decl_3_1;
```

We are now able to write a simplified version of the `LetInEnd` template class.

```
template<class T_decl, class T_exp>
struct LetInEnd
```

```
{
  template<class T_env>
  struct eval
     {
    typedef List<T_decl, T_env>              T_new_env;
    typedef T_exp::eval<T_new_env>::T        T;
    var_t doit()
    {
      // Create new variables declared in T_decl
      // (not detailed here).

      // Evaluate the expression in the new environment.
      var_t res = T_exp::eval<T_new_env>::doit();

      // Destroy the variables declared in T_decl
      // (not detailed here).

      return res;
    }
  };
};
```

All the needed operations on the environment can be done with type operations: we are able to fully compute the environment at compile-time for each evaluation point. Function calls are not detailed here, but they use the same operations. Note that all functions are evaluated each time they are called (as inline functions). This implies that if we want a recursive function to be translated as a C++ recursive function, we need the environment to be exactly the same at each recursive call.

6.5 The dynamic part

Not everything can be done at compile-time. The Tiger language allows some constructions which cannot be resolved statically.

The main dynamic stuff is the variable declaration and use. When a variable is declared, we need to store its value somewhere. At each evaluation point of the program, there is a set of variables which are accessible.

A variable can be of any supported Tiger type: it can be an array, a string or a record. There is no static representation of such values: we need to store this into memory at the program execution. Therefore we use the C++ stack: variables declared in a `let-in-end` construction are declared as local variables in the `LetInEnd` evaluation method.

The Tiger has a nested let declaration. At a given evaluation point, there can be several visible scopes. This obliges us to maintain a stack of scope pointers during the whole execution process. Accessing a variable is performed with two indirections: a first one to get the right scope and another one to reach the variable into this scope. We could have chose to implement variables access

with a static link mechanism. This would corresponded to the adaptable closure present in the phoenix library, part of the spirit project ([4]).

These indirections are our main limitation to really perform a static resolution of programs. Conversely, here is a program that is *entirely* statically evaluated:

Listing 4.5. A program solved statically

```
let
  function foo() = 20 * 20
  function bar() = 30 / 2
  function smousse() = if (80 > 6) then 1 else 0
in
   (foo() + bar() + smousse()) * 4
end
```

There is no variable used so, after our transform towards C++, we can expect that a C++ compiler can statically solve this program. In this particular case, using a C++ compiler which has good optimization capacities, we directly obtain one assembler instruction which gives the integer result.

6.6 The C++ program

The C++ program always have the same structure:

```
// Include all template classes needed to express and evaluate
   the AST.
#include "all.h"

// Generate the TAT.
typedef ... program_t;

int main()
{
  // instantiate program evaluation
  return (int)program_t::eval< initial_env_t >::doit();
}
```

The line of the main() launch the doit() instantiation, which results in the generation of the program evaluation code. This work is done by the C++ compiler.

7 Results

Our compiler covers all of the Tiger language. Lot of Tiger programs have been tested, and work successfully. Our process has been tested with como, g++ 3.2 and icc which gives slightly faster programs.

To experiment the performance of generated code, some Tiger programs compiled with our process have been compared to their C hard-coded equivalent. In

average, the C program goes two to three times faster than the (C++) Tiger one. This performance lack is mostly explained by the variable access cost: each access needs two indirections. But viewed as an evaluation process, this can be considered as good results.

This performance highly depends on the aptitude of the C++ compiler to optimize code. These optimizations are essentially obtained by the inlining mechanism. This optimization has been tested using the g++ option called `-finline-limit`. This option influences the quantity of functions inlined. This experience showed us the importance of good inlining at compile-time. Optimizations are done until approximatively `-finline-limit-1000`, which is much more than for usual C++ programs. This can be explained by the amount of functions that are instantiated. Indeed for each node of the AST, there is at least one function which will be used.

8 Conclusion

We have seen that a program can be expressed as an Abstract Syntax Tree (AST) given the language grammar. Using a technique based on Expression Templates, we are able to build a C++ type which describes this AST. This representation is called the TAT (Tree As Type).

Building and evaluating the TAT poses various problems. We need to express lists (for declarations, arguments, etc.). This problem is solved using the Static list technique. In the TAT, some elements refer to others. The reference problem implies the use of an environment which is implemented using a stack. We have seen that this container allows the required operations: pushing, popping and accessing. This stack is directly filled with parts of the TAT: this is a form of static pointers, which solves the reference problem.

An implementation based on the Tiger language has been proposed. This implementation intensively uses meta-programming techniques, therefore, the C++ compiler is able to do lot of work at compile-time: expression types and element references are solved statically. The limits of static resolution is the use of variables which can only be manipulated dynamically.

Our Tiger compiler is originally inspired by the Expression Templates technique. However, the evaluated constructions are not restricted to basic ones, such as unary or binary operators, but includes the common flow control constructions, structured types, variables, and nested functions. Moreover, thanks to the use of a static environment, such advanced operations can be evaluated by jumping from one point of the program to another. This happens for example each time a function is called. That characteristic is a noticeable difference with the Expression Templates which are evaluated in a simple bottom-up fashion.

This original technique shows how we used C++ meta-programming in order to work on abstract syntax trees of a mostly functional programming language. Indeed the C++ generative power allowed us to implement compiler parts and translation into C++ equivalent code.

References

[1] A.W. Appel. *Modern Compiler Implementation in C / Java / ML.* Cambridge University Press, 1997.

[2] J.A. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T.J. Williams. Generic programming in POOMA and PETE. In *Generic Programming, Proceedings of the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 218–. Springer-Verlag, 2000.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[4] Spirit group. Spirit parser framework, 2002.

[5] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science and Engineering*, 1(4), 1999.

[6] G. Powell J. Jarvi. The boost lambda library, 2002.

[7] J. Järvi. Compile time recursive objects in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–77. IEEE Computer Society Press, 1998.

[8] Brian McNamara and Yannis Smaragdakis. Functional programming in C++ using the FC++ library. *SIGPLAN Notices*, April 2001.

[9] Jörg Striegnitz and Stephen A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[10] C. van Reeuwijk. Rapid and robust compiler construction using template-based metacompilation. In *12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 247–, Warsaw, Poland, April 2003. Springer-Verlag.

[11] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[12] T. Veldhuizen. Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA, 2002.

A A simple C++ meta-program and its evaluation

```
template<unsigned i>
struct factorial
{
  enum {res = i * factorial< i - 1 >::res};
};

template<>
struct factorial<0>
{
  enum {res = 1};
};

enum { fact4 = factorial<4>::res };
```

Thanks to the template expansion mechanism, this C++ meta-function allows to compute a factorial at compile-time:

```
factorial<4>::res
4 * factorial<3>::res
4 * 3 * factorial<2>::res
   4 * 3 * 2 * factorial<1>::res
4 * 3 * 2 * 1 * factorial<0>::res
4 * 3 * 2 * 1 * 1
24
```

B A full tiger program

```
let
 type any = {any : int}
 var buffer := getchar()

 function printint(i: int) =
  let function f(i:int) = if i>0
             then (f(i/10); print(chr(i-i/10*10+ord("0"))))
   in if i<0 then (print("-"); f(-i))
      else if i>0 then f(i)
      else print("0")
  end


function readint(any: any) : int =
 let var i := 0
     function isdigit(s : string) : int =
                  ord(buffer)>=ord("0") & ord(buffer)<=ord
                    ("9")
     function skipto() =
       while buffer=" " | buffer="\n"
         do buffer := getchar()
  in skipto();
     any.any := isdigit(buffer);
     while isdigit(buffer)
       do (i := i*10+ord(buffer)-ord("0"); buffer := getchar()
         );
     i
 end

 type list = {first: int, rest: list}

 function readlist() : list =
    let var any := any{any=0}
        var i := readint(any)
     in if any.any
         then list{first=i,rest=readlist()}
         else nil
```

```
    end

 function merge(a: list , b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first ,rest=merge(a.rest ,b)}
      else list{first=b.first ,rest=merge(a,b.rest)}

 function printlist(l: list) =
   if l=nil then print("\n")
   else (printint(l.first); print(" "); printlist(l.rest))

   var list1 := readlist()
   var list2 := (buffer:=getchar(); readlist())

 in
        print ("list 1 : \n");
        printlist (list1);
        print ("list 2 : \n");
        printlist (list2);
        print ("merged list : \n");
        printlist (merge (list1 , list2) )
end
```

C TAT of the previous program

The following program compiles in less than two minutes with g++ 3.2 on a 350Mhz processor.

```
#include "all.h"

typedef LetInEnd< List< RecordType< List< TypeLnk< builtin_types
  , 1 > > > >,
LetInEnd< List< Variable< FuncCall< builtin_funcs , 9 , List< > >,
  builtin_types , 2 > >,
LetInEnd< List<
Function< List< TypeLnk< builtin_types , 1 > >, LetInEnd< List<
Function< List< TypeLnk< builtin_types , 1 > >, If< BinOp< SimpleVar
  < 5, 0 >, ConstInt< 0 >, GreatThan >, ExpList< FuncCall< 4, 0, List<
  BinOp< SimpleVar< 5, 0 >, ConstInt< 10 >, Divide > > >, ExpList<
  FuncCall< builtin_funcs , 0 , List< FuncCall< builtin_funcs , 4 , List<
  BinOp< BinOp< SimpleVar< 5, 0 >, BinOp< BinOp< SimpleVar< 5, 0 >,
  ConstInt< 10 >, Divide >, ConstInt< 10 >, Times >, Minus >, FuncCall<
  builtin_funcs , 3 , List< ConstString< 0 > > >, Plus
   > > > > > > > > >, 4 >
 >,
ExpList< If< BinOp< SimpleVar< 3, 0 >, ConstInt< 0 >, LessThan >,
  ExpList< FuncCall< builtin_funcs , 0 , List< ConstString< 1 > > >,
  ExpList< FuncCall< 4, 0, List< BinOp< ConstInt< 0 >, SimpleVar
  < 3, 0 >, Minus > > > >, If< BinOp< SimpleVar< 3, 0 >, ConstInt
  < 0 >, GreatThan >, FuncCall< 4, 0, List< SimpleVar< 3, 0 > > >,
  FuncCall< builtin_funcs , 0 , List< ConstString< 2 > > > > > > >
, 2 >
, List<
```

```
Function< List < TypeLnk< 0, 0 > >, LetInEnd< List < Variable < ConstInt
  < 0 >, builtin_types , 1 > >,
   LetInEnd< List <
Function< List < TypeLnk< builtin_types , 2 > >, If < BinOp< FuncCall <
  builtin_funcs , 3 , List < SimpleVar < 1, 0 > > >, FuncCall < builtin_funcs
  , 3 , List < ConstString < 3 > > >, GreatEq >, BinOp< FuncCall <
  builtin_funcs , 3 , List < SimpleVar < 1, 0 > > >, FuncCall < builtin_funcs
  , 3 , List < ConstString < 4 > > >, LessEq >, ConstInt < 0 > >, 5 >
, List <
Function< List <  >, While< If < BinOp< SimpleVar < 1, 0 >, ConstString
  < 5 >, Equal >, ConstInt < 1 >, BinOp< SimpleVar < 1, 0 >, ConstString
  < 6 >, Equal > >, Assign < SimpleVar < 1, 0 >, FuncCall < builtin_funcs
  , 9 , List <  > > > >, 5 >
 > >,
ExpList< FuncCall < 5 , 1 , List <  > >, ExpList< Assign < FieldVar <
  SimpleVar < 3 , 0 > , 0 >, FuncCall < 5 , 0 , List < SimpleVar
  < 1, 0 > > > >, ExpList< While< FuncCall < 5 , 0 , List < SimpleVar
  < 1, 0 > > >, ExpList< Assign < SimpleVar < 4 , 0 >, BinOp< BinOp< BinOp
  < SimpleVar < 4 , 0 >, ConstInt < 10 >, Times >, FuncCall < builtin_funcs
  , 3 , List < SimpleVar < 1, 0 > > >, Plus >, FuncCall < builtin_funcs , 3 ,
  List < ConstString < 7 > > >, Minus > >, ExpList< Assign < SimpleVar
  < 1, 0 >, FuncCall < builtin_funcs , 9 , List <  > > > > > >, ExpList<
  SimpleVar < 4, 0 > > > > > > >
, 2 >
 > >,
LetInEnd< List < RecordType< List < TypeLnk< builtin_types , 1 >, List <
  TypeLnk< 3, 0 > > > > >,
LetInEnd< List <
Function< List <  >, LetInEnd< List < Variable < Record < 0 , 0 , List <
  ConstInt < 0 > > >, 0, 0 >, List < Variable < FuncCall < 2 , 1 , List <
  SimpleVar < 6, 0 > > >, builtin_types , 1 > > >,
ExpList< If < FieldVar < SimpleVar < 6, 0 >, 0 >, Record < 3 , 0 , List <
  SimpleVar < 6, 1 >, List < FuncCall < 4 , 0 , List <  > > > > >, Nil > > >
, 4 >
, List <
Function< List < TypeLnk< 3 , 0 >, List < TypeLnk< 3 , 0 > > >, If < BinOp<
  SimpleVar < 5 , 0 >, Nil , Equal >, SimpleVar < 5 , 1 >, If < BinOp<
  SimpleVar < 5 , 1 >, Nil , Equal >, SimpleVar < 5 , 0 >, If < BinOp<
  FieldVar < SimpleVar < 5 , 0 >, 0 >, FieldVar < SimpleVar < 5 , 1 >, 0 >,
  LessThan >, Record < 3 , 0 , List < FieldVar < SimpleVar < 5 , 0 >, 0 >, List
  < FuncCall < 4 , 1 , List < FieldVar < SimpleVar < 5 , 0 >, 1 >, List <
  SimpleVar < 5, 1 > > > > > > >, Record < 3 , 0 , List < FieldVar < SimpleVar
  < 5 , 1 >, 0 >, List < FuncCall < 4 , 1 , List < SimpleVar < 5 , 0 >, List <
  FieldVar < SimpleVar < 5, 1 >, 1 > > > > > > > > > > >, 4 >
, List <
Function< List < TypeLnk< 3 , 0 > >, If < BinOp< SimpleVar < 5 , 0 >, Nil ,
  Equal >, FuncCall < builtin_funcs , 0 , List < ConstString < 8 > > >,
  ExpList< FuncCall < 2 , 0 , List < FieldVar < SimpleVar < 5 , 0 >, 0 > > >,
  ExpList< FuncCall < builtin_funcs , 0 , List < ConstString < 9 > > >,
  ExpList< FuncCall < 4 , 2 , List < FieldVar < SimpleVar
  < 5 , 0 >, 1 > > > > > > >, 4 >
 > > >,
LetInEnd< List < Variable < FuncCall < 4 , 0 , List <  > >, builtin_types
  , 0 >, List < Variable < ExpList< Assign < SimpleVar < 1, 0 >, FuncCall <
  builtin_funcs , 9 , List <  > > >, ExpList< FuncCall < 4 , 0 , List <
   > > > >, builtin_types , 0 > > >,
ExpList< FuncCall < builtin_funcs , 0 , List < ConstString < 10 > > >,
  ExpList< FuncCall < 4 , 2 , List < SimpleVar < 5 , 0 > > >, ExpList<
  FuncCall < builtin_funcs , 0 , List < ConstString < 11 > > >, ExpList<
  FuncCall < 4 , 2 , List < SimpleVar < 5 , 1 > > >, ExpList< FuncCall <
  builtin_funcs , 0 , List < ConstString < 12 > > >, ExpList< FuncCall
  < 4 , 2 , List < FuncCall < 4 , 1 , List < SimpleVar < 5 , 0 >, List < SimpleVar
  < 5, 1 > > > > > > > > > > > > > > > > > > > > >

        program_t;
```

```
const char*      metasmousse::const_string[] = {"0", "-", "0", "0", "9",
  "␣", "\012", "0", "\012", "␣", "list␣1␣:␣\012", "list␣2␣:␣\012", "
  merged␣list␣:␣\012", NULL};

int main()
{
  return (int)program_t::eval< initial_env_t >::doit();
}
```

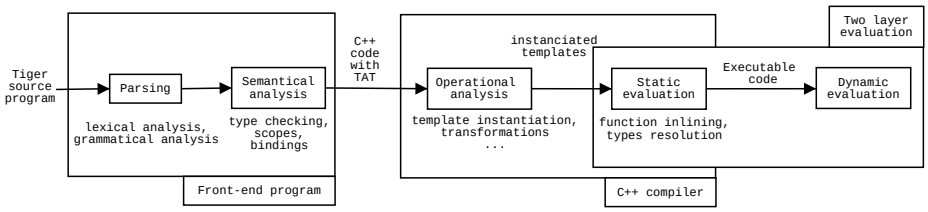

Fig. 1. Placement in the compilation chain

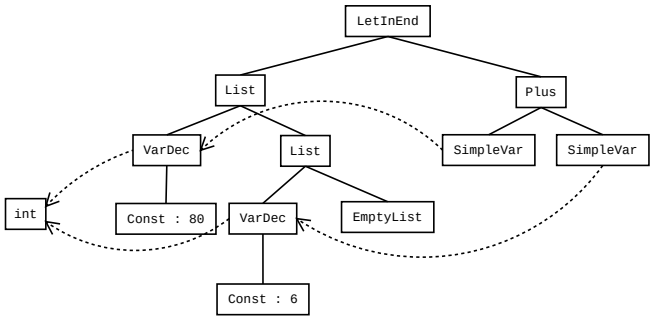

Fig. 2. AST of example 4.3

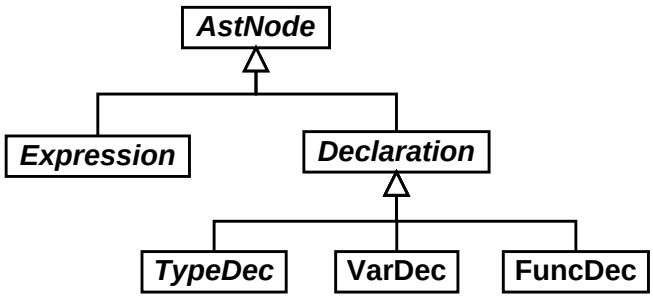

Fig. 3. Main kind of classes

JSetL: Declarative Programming in Java with Sets

Elisabetta Poleo and Gianfranco Rossi

Dip. di Matematica, Università di Parma, Dip. di Matematica, Via M. D'Azeglio 85/A, 43100 Parma (Italy). gianfr@prmat.math.unipr.it

Abstract. In this paper we present a Java library—called JSetL—that offers a number of facilities to support declarative programming like those usually found in CLP languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism. The paper describes the main features of JSetL and it shows how these features can be exploited to write in Java declarative solutions to a number of simple problems in a CLP style.

1 Introduction

Many different constraint solvers have been designed in the last twenty years, devoted to different constraint domains with different features, and using different implementation techniques. Most of them have been proposed in the context of Constraint Logic Programming (CLP). Among them we can mention Prolog III and IV [5, 16], CHIP [10, 19], CLP(R) [14], GNU Prolog (formerly clp(FD)) [7, 3], CLP(SET) [8] and ECLIPSE [11].

On the other hand, from the beginning of '90ties, researchers have realized that it can be convenient to have constraint solvers embedded in a more conventional programming setting (in particular an object-oriented one), in which one can accommodate the fundamental capabilities of CLP as well as those constructs for programming and software structuring that are typical of conventional programming languages. As a matter of fact, most real-world software development is done using conventional programming languages, in particular, object-oriented languages such as C++ and Java.

Among the proposals that move along these lines one of the best known is that of the ILOG Solver [15, 13]. In this system, constraints and logical variables are handled as objects and are defined within a C++ class library. Thanks to the encapsulation and operator overloading mechanisms, programmers can view constraints almost as if they really were part of the language. Similar proposals are those of INC++ [12], NeMo+ [18], and JSolver [4], the last one based on the Java language instead of on C++.

In all these proposals the constraint solvers are viewed as *libraries* of the host language, more or less integrated with the language itself. A different approach for allowing constraints in a conventional programming language consists in defining a new programming language, or extending an existing one, in such

a way constraints are viewed as "first-class citizens" of the language itself. This is the solution adopted for instance in the languages Alma-0 [2], Singleton [17], and DJ (Declarative Java) [20, 21].

A potential advantage of this approach with respect to that based on a library is that it allows a tighter integration between constructs of the host language and the facilities offered by the constraint solver, making programs simpler and more "natural" to write. On the other hand, however, the design and development of a new language is surely a more difficult task, and the resulting systems are likely to be less easy to integrate with other existing systems.

The work presented in this paper is another proposal following the OO library approach: we endow an OO language, namely Java, with facilities for defining and manipulating constraints, by providing them as a library—called JSetL. What is peculiar in our proposal, however, is the kind of data abstractions and constraints that the library provides, and the programming style that these facilities support. Specifically, some notable features of JSetL are:

- logical variables;
- list and set data structures, possibly partially specified (i.e., containing uninitialized logical variables)
- unification (in particular, unification over lists and sets)
- a powerful set constraint solver which allows to compute with partially specified data
- nondeterminism (though confined to constraint solving).

These facilities provide a valuable support to *declarative programming*. In particular the constraint solver allows complex (set) expressions to be checked for satisfiability on a specific domain, disregarding the order in which they are encountered and the instantiation of variables occurring in them. Moreover, the use of partially specified set data structures, along with the nondeterminism "naturally" supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modelling tool, in particular for combinatorial problems.

All the features listed above for JSetL are present also in the CLP($\mathcal{SET}$) language [8], but embedded in a CLP framework. An attempt to "export" these features outside CLP is represented by the definition of the SINGLETON language [17], a declarative language that combines most of the features considered in this paper with "traditional" features of imperative programming languages, such as the iterative control structures and the block structure of programs. SINGLETON, however, is a completely new language, with its own syntax and its own semantics. One of the aims of this work is to allow us to compare the approach followed in SINGLETON with the library based approach followed in JSetL, in order to evaluate the gain in the expressive power related to the effort needed to develop the new facilities and the easiness to use them. Actually, the debate about pros and cons of the two approaches is still largely open.

The paper is organized as follows. In Section 2 we give an informal presentation of JSetL by showing a simple Java program using JSetL. In Section 3 we

introduce the fundamental data structures of JSetL, namely logical variables, sets and lists. In Section 4, we describe the (set) constraint handling facilities supported by our library and we show how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. The fundamental notion of nondeterminism and its relationship with sets is also addressed in this section. In Section 5 we show how user defined constraints can be introduced in a program and how they can be used. Finally, in Section 6 we briefly discuss future work.

2 An informal introduction to programming with JSetL

First of all we show a simple example of a Java program using JSetL which allows us to give the flavor of the programming style supported by the library.

Problem: Compute and print the maximum of a set of integers.

For the sake of simplicity we assume that the set of integers is directly supplied by the program (instead of being read for instance from a file). Hence we will focus on the definition of the method `max` that computes the maximum of a set `s` of integers. Observe that the proposed implementation does not take care of execution efficiency. Indeed, JSetL is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

```
class Max
    {
    public static Lvar max(Set s) throws Failure
        {
        Lvar x = new Lvar();
        Lvar y = new Lvar();
        Solver.add(x.in(s));
        Solver.forall(y,s,y.leq(x));
        Solver.solve();
        return x;
        }

    public static void main (String[] args) throws IOException, Failure
        {
        int[] sample_set_elems = {1,6,4,8,10,5};
        Set sample_set = new Set(sample_set_elems);
        System.out.print(" Max = "); max(sample_set).print();
        }
    }
```

`x` and `y` are two logical variables and both are uninitialized. Invocation of the `add` method adds the constraint `x.in(s)` (i.e., $x \in s$) to the current constraint store. This constraint is evaluated to `true` if `s` is a set and `x` belongs to `s`. If `x` is uninitialized when the expression is evaluated this amounts to *nondeterministically* assign an element of `s` to `x`. Invocation of the `forall` method allows to

add to the constraint store a new constraint `y.leq(x)` (i.e., `y` $\leq$ `x`) for each `y` belonging to `s`. As soon as the `solve` method is invoked the constraint solver checks whether the current collection of constraints in the constraint store is satisfiable or not. If it is, the invocation of the `solve` method terminates with success. The value of `x` represents the integer we are looking for and it is returned as the result of `max`. If, on the contrary, one of the constraints in the constraint store is evaluated to false, backtracking takes place and the computation goes back till the nearest choice point. In this case, the nearest and only choice point is the one created by the `x.in(s)` constraint. Its execution will bind nondeterministically `x` to each element of `s`, one after the other. If all values of `s` have been attempted, there is no further alternative to explore and the computation of `max` terminates raising an exception `Failure`. If no **catch** clause for this exception is provided, the whole computation terminates reporting a failure (actually this is not the case of the `max` method, since a value of `x` for which all the constraints hold surely exists—exactly the maximum of `s`).

Executing the program with the sample set of integers declared in the **main** method causes the message `Max = 10` to be printed to the standard output.

3 Logical variables and composite data objects

JSetL provides logical variables and two new kinds of data structures: sets and lists. These new features are implemented by three classes, `Lvar`, `Lst`, and `Set`, for creation and manipulation of logical variables, lists and sets, respectively.

A *logical variable* is an instance of the class `Lvar`, created by the statement

```
Lvar VarName = new Lvar(VarNameExt,VarValue);
```

where VarName is the variable name, VarNameExt is an optional *external name* of the variable, and VarValue is an optional ***Lvar*** *value* associated with the variable.

The external name is a string value which can be useful when printing the variable and the possible constraints involving it (if omitted, a default name of the form `"Lvar_`n`"`, where n is a unique integer, is assigned to the variable automatically). An `Lvar` value can be either a primitive type value, or any library or user defined class object (provided it supplies a method `equals` for testing equality between two instances of the class itself). In particular, an `Lvar` value can be an instance of `Lvar`, `Lst`, or `Set`

A logical variable which has no `Lvar` value associated with it or whose `Lvar` value is an uninitialized logical variable (or list or set), is said to be *uninitialized* (or an *unknown*). Otherwise, the logical variable is *initialized*. `Lvar` values other than uninitialized logical variables (or lists or sets) are said *known values*. Uninitialized logical variables will possibly assume a known value (i.e., they become initialized) during the computation, in consequence of some constraints involving them.

A *list* is a finite (possibly empty) sequence of arbitrary `Lvar` values (i.e., the *elements* of the list). In JSetL a list is an instance of the class `Lst`, created by the statement

```
Lst LstName = new Lst(LstNameExt,LstElemValues);
```

where LstName is the list name, LstNameExt is an optional *external name* of the list, and VarElemValues is an optional array of `Lvar` values $c_1, \ldots, c_n$ of type t, which constitute the elements of the list. The constant `Lst.empty` is used to denote the *empty list*.

A list can be either initialized or uninitialized. An uninitialized list is like a logical variable, but constrained to be (possibly) initialized by list objects only.

Hereafter, we will often make use of an abstract notation—which closely resembles that of Prolog—to write lists in a more convenient way. Specifically, $[e_1, e_2, \ldots, e_n]$ is used to denote the list containing n elements e_1, e_2, ..., e_n, while $[\,]$ is used to denote the *empty list*. Moreover, $[e_1, e_2, \ldots, e_n \mid R]$, where R is a list, is used to denote a list containing the n elements e_1, e_2, ..., e_n, plus elements in R. In particular, if R is uninitialized, $[e_1, e_2, \ldots, e_n \mid R]$ represents an "unbound" list, with elements e_1, ..., e_n and an unknown part R. Similar abstract notation will be introduced also to represent sets (with square brackets replaced by curly brackets).

A *set* is a finite (possibly empty) collection of arbitrary `Lvar` values (i.e., the *elements* of the list). While in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. In JSetL a set is an instance of the class `Set`, created by the statement

```
Set SetName = new Set(SetNameExt,SetElemValues);
```

where SetName, SetNameExt, and SetElemValues have the same meaning than in lists. The constant `Set.empty` is used to denote the *empty set*. Also, a set can be either initialized or uninitialized.

Example 1 . `Lvar`, `Lst`, *and* `Set` *definitions*

```
Lvar x = new Lvar();          // uninitialized l. var.
Lvar y = new Lvar("y",'a');   // initialized l. var. (value 'a'),
                              // with ext'l name "y"
Lvar t = new Lvar(x);         // uninitialized l. var.;
                              // same as variable x
Lst l = new Lst("l");         // uninitialized list,
                              // with ext'l name "l"
int[] s_elems = {2,4,8,3};
Set s = new Set("s",s_elems); // initialized set (value {2,4,8,3}),
                              // with ext'l name "s"
```

Elements of a list or of a set can be also logical variables (or lists or sets), possibly uninitialized. For example, the following declarations

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Lst pl = new Lst(pl_elems);
```

create the list `pl` with value `[1,x]`, where `x` is an uninitialized logical variable. A list (resp., set) that contains some elements which are uninitialized logical variables (or lists, or sets) is said a *partially specified list (set)*. Note that in a

partially specified set the cardinality is not completely determined. For example, the partially specified set {1,x} has cardinality 1 or 2 depending whether x will get value 1 or different from 1, respectively (actually, each partially specified set/list denotes a possibly infinite collection of different sets/lists, that is all sets/lists which can be obtained by assigning admissible values to the uninitialized variables).

A list (resp., set) can be also obtained as the result of evaluating a list (resp., set) constructor expression. Let e be an ***Lvar*** *expression* (i.e. an expression returning a Lvar value), l and m be *list expressions* (i.e., expressions returning a list object or a logical variable whose value is a list object), and x be an uninstantiated logical variable. A *list constructor* is an expression of one of the forms:

(i) `l.ins1(`e`)` (head element insertion) (iii) `l.ext1(x)` (head element removal)
(ii) `l.insn(`e`)` (tail element insertion) (iv) `l.extn(x)` (tail element removal)

Expressions (i) and (ii) denote the list obtained by adding $val(e)$ as the first and the last element of the list l, respectively, whereas expressions (iii) and (iv) denote the list obtained by removing from l the first and the last element, respectively. Evaluation of expressions (iii) and (iv) also causes the value of the removed element to become the value of x. [1]

It is important to notice that these methods do not modify the list on which they are invoked: rather they build and return a new list obtained by adding/removing the elements to/from the input list (the same will hold for sets, too).

Constructor expressions for sets are simpler than those for lists. In fact, in lists we can distinguish between the first (the *head*) and the last (the *tail*) element of a list, while in sets the order of elements is immaterial. Moreover, only the element insertion method is provided since element extraction may involve a non-deterministic selection of the element to be extracted that is better handled using set constraints (see Section 4).

Let e be an Lvar expression and s be a *set expression* (i.e., an expression returning a set object or a logical variable whose value is a set object). A *set constructor* is an expression of the form:

`s.ins(`e`)` (element insertion)

which denotes the set obtained by adding $val(e)$ to s (i.e., $\mathtt{s} \cup \{val(e)\}$).

Set/List insertion and extraction methods can be concatenated (left associative). In fact these methods always return a Set/Lst object, and the returned object can be used as the invocation object as well.

Using the insertion methods it is also possible to build *unbounded* partially specified sets/lists, that is data structures with a certain number of (either known or unknown) elements $e_1, ..., e_n$, and an unknown "rest" part, represented by

[1] Extraction methods for lists require that the invocation list l is initialized and that x is not initialized. If one of these conditions is not respected an exception is raised (namely, NotInitVarException and InitLvarException, respectively). Moreover, if l is the empty list, a EmptyLstException exception is raised.

an uninitialized set/list r (i.e., using the abstract notation, $\{e_1, \ldots, e_n \mid r\}$ or $[e_1, \ldots, e_n \mid r]$ for sets and lists, respectively).

Example 2 . *Set/List element insertion and removal*

```
Lvar nil = new Lvar(Lst.empty);                 // the empty list
Lst l1 = new Lst(nil.ins1(3+2).ins1(x));        // the p.s. list [x,5]
                                                // (x uninitialized var.)
Lst l2 = new Lst(l1.ext1(y).insn(y));           // the p.s. list [5,x]
Set s1 = new Set(Set.empty.ins(1).ins('a')); // the set {'a',1}
Set r = new Set();                              // an uninitialized set
Set s2 = new Set(r.ins(1));                     // the unbounded set {1 | r}
```

Note that `s2` in the above example is a partially specified set containing one element, `1`, and an unknown part `r`; in this case, the cardinality of the denoted set has no upper bound (the lower being 1).

Special forms of the insertion and extraction methods are provided to simplify their usage. In particular, the method `ins1All(a)`, applied to a list `l`, where `a` is an array of elements of a type t, returns a list obtained from `l` by adding all elements of `a` as the head elements of `l`, respecting the order they have in `a`. Similarly, `insAll(a)`, applied to a set `s`, is used to insert more than one element at a time into `s`. In addition, an alternative form is provided for specifying the value for a set or list object. When creating the object it is possible to specify the limits l and u of an interval $[l, u]$ of integers: the elements of the interval will be the elements of the set/list (if $u < l$ the set/list is empty).

A number of utility methods are also provided by classes `Lvar`, `Lst`, and `Set`. These methods are used, for example, to print a set/list object, to know whether a logical variable is initialized or not, to get the external name associated with a `Lvar`, `Lst`, or `Set` object, and so on.

Logical variables, sets, and lists are used mainly in conjunction with constraints. Constraints are addressed in more details in the next section.

4 (Set) Constraints

Basic set-theoretical operations, as well as equalities and disequalities, are dealt with as *constraints* in JSetL. The evaluation of expressions containing such operations is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the global *constraint store*) using domain specific constraint solvers. Those parts of these expressions, usually involving one or more uninitialized variables, which cannot be completely solved are added to the constraint store and will be used to narrow the set of possible values that can be assigned to the uninitialized variables.

The approach adopted for constraint solving in JSetL is the one developed for CLP($\mathcal{SET}$)[8]. Logically, the constraint store is a conjunction of atomic formulae built using basic set-theoretic operators, along with equality and disequality. Satisfiability is checked in a set-theoretic domain, using a suitable constraint

solver which tries to reduce any conjunction of constraints to a simplified form—the *solved form*—which can be easily tested for satisfiability. The success of this reduction process allows one to conclude the satisfiability of the original collection of constraints. Conversely, the detection of a failure (logically, the reduction to false) implies the unsatisfiability of the original constraints. Solved form constraints are left in the current constraint store and passed ahead to the new state. A successful computation, therefore, may terminate with a not empty collection of solved form constraints in the final constraint store.

An *atomic constraint* in JSetL is an expression of one of the forms:

- $e_1.op\,(e_2)$
- $e_1.op\,(e_2, e_3)$

where e_1 is either a `Lvar`, a `Lst` or a `Set` expression, e_2 and e_3 can be `Lvar`, `Lst` or `Set` expressions, a primitive type value or any class object provided of an `equal` method. *op* is one of a collection of predefined operators that implement basic operations on sets, such as: equality, membership, (strict) inclusion, union, disjunction, intersection, set difference, and, for most of them, also their negative counterparts. In particular, set equality turns out to be dealt with as a *set unification* problem [9].

A *constraint* is the conjunction of two or more atomic constraints v_1, v_2,...,v_n:

- v_1.**and**(v_2) ... .**and**(v_n)

Example 3 . *Set constraints*

Let `x`, `y`, `z` *be logical variables and* `r`, `s`, *and* `t` *be sets.*

```
r.eq(s));                                  // equality between sets
t.union(r,s));                             // t = r ∪ s
x.eq(y).and(x.eq(3)).and(y.neq(z)))        // x = y ∧ x = 3 ∧ x ≠ z
```

A constraint C can be added to the constraint store by calling the `add` method of the `Solver` class

```
Solver.add(C)
```

The order in which constraints are added to the constraint store is completely immaterial. After constraints have been added to the store, one can invoke their resolution by calling the `solve` method:

```
Solver.solve()
```

The `solve` method nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. If there is no solution a `Failure` exception is generated. We say that the invocation of a method, calling (directly or indirectly) the `solve` method, terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred.

To find a solution, the constraint solver tries to reduce the atomic constraints in the constraint store to a simplified form - called the *solved form* (see [8]). This reduction is *nondeterministic*. Nondeterminism is handled through choice points and backtracking. Once the constraint reduction process detects a failure, the computation backtracks to the most recently created choice point (chronological backtracking). If no choice point is left open the whole reduction process fails (i.e., the `Failure` exception is generated).

Example 4 . *Constraint solving*

Let `s` *be the set* {`x,y,z`}*, where* `x`*,* `y`*, and* `z` *are uninitialized logical variables, and* `r` *be the set* {`1,2,3`}*.*

```
Solver.add(r.eq(s));       // set unification r = s
Solver.add(x.neq(1));      // x ≠ 1
Solver.solve();            // calling the constraint solver
x.output();
```

`x.output()` *prints the (external) name of the variable* `x` *followed by its value (if any; otherwise, followed by* `'_'`*). Therefore the output generated by this code fragment is:*

```
x = 2
```

The value for `x` is computed through backtracking; as a matter of fact, the first value for `x` computed by solving `r.eq(s)` is (likely to be) `1`, which however does not satisfy the other constraint `x.neq(1)`. Thus, backtracking forces the solver to find another solution for `x`, namely `x = 2`. In this case, the conjunction of the two given constraints is satisfied, and the invocation of the `solve` methods terminates successfully.

If later on a new constraint, e.g., `[x]` $\neq$ `[2]`, is added to the constraint store, and the constraint solver is called again, the choice points left open by the previous call to the solver are still open and they are explored by the new invocation.

```
Solver.add(Lst.empty.ins1(x).neq(Lst.empty.ins1(2))); // [x] ≠ [2]
Solver.solve();
x.output();
```

The output generated at the end of the computation of this new fragment of code is therefore:

```
x = 3
```

Note that every time the `solve` method is invoked it does not restart solving the constraint from the beginning but it restart from the point reached by the last invocation to `solve`.

At the end of the computation the constraint store may contain solved form constraints. To print these constraints, other than equality constraints, one can use the static method `showStore()` of class `Solver` (actually this method allows

to visualize the content of the constraint store at any moment during the computation). Let us see how the solver works on a simple example involving also negative constraints in the computed result.

Example 5 *Programming with constraints.*

Check whether an element `x` *belongs to the difference between two sets,* `s1` *and* `s2` *(i.e.,* `x` $\in$ `s1`$\backslash$`s2`*)* .

```
public static void in_difference(Lvar x, Set s1, Set s2) throws Failure
    {
    Solver.add(x.in(s1));
    Solver.add(x.nin(s2));
    }
```

If the following code fragment is executed (for instance, in the **main** *method)*

```
in_difference(v,s,r);
x.output();
Solver.showStore();
```

and `s` *and* `r` *are the sets* {`1,2`} *and* {`1,3`}*, respectively, and* `x` *is an uninitialized variable, the output generated is:*

```
x = 2
Store: empty
```

Conversely, if `s` *is an uninitialized set, then executing the same program fragment as above, will produce the following output*

```
x = _
s = {x | Set_1}
Store: x.neq(1) x.neq(3)
```

which is read as: `s` *can be any set containing the element* `x` *and* `x` *must be different from* `1` *and* `3`.

The ability to solve constraints disregarding the fact logical variables occurring in them are initialized or not allows methods involving constraints to be used in a quite flexible way, e.g., using the same method both for testing and computing solutions This flexibility strongly contributes to support a declarative programming style.

A convenient way to introduce more than one constraint at a time is by using the `forall` method. Let `x` be an uninitialized variable, S a set expression which is evaluated to an initialized set, C a constraint containing `x`, and C_s the constraint obtained from C by replacing all occurrences of `x` with element s of S. The statement

`Solver.forall(x,`S`,`C`)`

adds the constraint C_s to the constraint store, for each element s of S. Logically, `forall(x,`S`,`C`)` is the so-called Restricted Universal Quantifier: $\forall x((x \in S) \rightarrow C)$ (see the sample program in Section 2 for a simple use of `forall`).

It is common also to allow *local* variables $y_1, \ldots, y_n$ in C, which are created as new for each element of the set (logically, $\forall x((x \in S) \rightarrow \exists y_1, \ldots, y_n(C))$ that is $y_1, \ldots, y_n$ are existentially quantified variables). For this purpose, JSetL provides also the method

`Solver.forall(x,`S`,`Y`,`C`)`

where `x`, S, and C are the same as in the simpler `forall` method, while Y is a list of uninitialized logical variables.

Example 6 *Using the* `forall` *method.*

Check whether all elements of a set `s` *are pairs, i.e., they have the form* `{x1,x2}`*, for any* `x1` *and* `x2`*.*

```
public static void all_pairs(Set s) throws Failure
    {
    Lvar x1 = new Lvar();
    Lvar x2 = new Lvar();
    Lst Y = new Lst(Lst.empty.ins1(x2).ins1(x1));
    Lvar x = new Lvar();
    Solver.forall(x,s,Y,x.eq(Lst.empty.ins1(x2).ins1(x1)));
    Solver.solve();
    return;
    }
```

Let `sample_set` *be the set* `{[1,3],[1,2],[2,3]}`*. The following fragment of code tests whether* `sample_set` *is composed only of pairs and prints a message* ``` ``All pairs'' ``` *or* ``` ``Not all pairs'' ``` *depending on the result of the test.*

```
boolean res = true;
try {
    all_pairs(sample_set);
    }
catch(Failure e)
    {res = false;}
if (res) System.out.print("All pairs");
else System.out.print("Not all pairs");
```

Example 6 shows also how a statement, namely the call `all_pairs(sample_set)`, can be used, in a sense, as a condition. In fact, if execution of the statement fails (i.e., not all elements in the given set are pairs), then an exception `Failure` is raised and the associated exception handler executed. The latter can easily set a boolean variable to be used in the next **if** statement. Thus, if the statement terminates with success then a `true` value is returned (in `res`); otherwise, the statement terminates with failure and a `false` value is returned. This is analogous to the use of statements as expressions found in some languages, such as Alma-0 [1]) and SINGLETON [17].

A computation in JSetL can be nondeterministic, though nondeterminism in JSetL is confined to constraint solving. Precisely, like in SINGLETON, nondeterminism is mainly supported by set operations. As a matter of fact, the

notion of nondeterminism fits into that of set very naturally. Set unification and many other set operations are inherently and naturally nondeterministic. For example, the evaluation of $x \in \{1, 2, 3\}$ with x an uninitialized variable, nondeterministically returns one among $x = 1$, $x = 2$, $x = 3$. Since the semantics of set operations is usually well understood and quite "intuitive", making nondeterministic programming the same as programming with sets can contribute to make the (not trivial) notion of nondeterminism easier to understand and to use.

Nondeterminism is another key feature of a programming language to support declarative programming.

A simple way to exploit nondeterminism in JSetL is through the use of the `Setof` method. This method allows one to explore the whole search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`. Then the collected set can be processed, e.g., by iterating over all its elements using the `forall` method.

Example 7 *All solutions.*

Compute the set of all subsets (i.e., the powerset) of a given set `s`.

```
public static Set powerset(Set s) throws Failure
    {
    Set r = new Set();
    Solver.add(r.subset(s));
    Solver.setof(r);
    return r;
    }
```

If `s` *is the set* `{'a','b'}`*, the set returned by* `powerset` *is* `{{},{'a'},{'b'},{'a','b'}}`.

Constraints and other JSetL facilities can be used in conjunction with the usual control structures of Java. This situation is illustrated by the following example.

Example 8 *Symmetrical list.*

Check whether a list `l` *is symmetrical or not.*

```
public static boolean symmetrical(Lst l) throws Failure
    {
    try {
        while(l.size()>1)
            {
            Lvar z1 = new Lvar();
            Lvar z2 = new Lvar();
            Lst r = l.ext1(z1).extn(z2); // extract the first and last element of l
            Solver.add(z1.eq(z2)); // the first and the last must be equal
            Solver.solve();
```

```
            l = r; // continue with the rest of l
            }
        return true;
        }
    catch(Failure e)
        {
        return false;
        }
}
```

If, for example, `l` *is* `['r','a','d','a','r']` *the value returned by* `symmetrical` *is* **true**. *List* `l` *can contain also some unknown values. For example, with* `l = [x,1,3,y,2]`, `x` *and* `y` *uninitialized variables, invocation of the* `symmetrical` *method returns* **true** *and as a side-effect it initializes variables* `x` *and* `y` *to* `2` *and* `1`, *respectively. Note that within the* **while** *loop we use an assignment between two logical variables,* `l = r`: *this forces* `l` *at the next iteration to be replaced by the new (shorter) list* `r`.

Finally we show the application of JSetL to a more complex problem, the well-known combinatorial problem of the coloring of a map.

Example 9 *Coloring of a map.*

Given a map of n regions r_1,...,r_n *and a set of m colors* c_1,...c_m *find an assignment of colors to regions such that neighboring regions have different colors.*

The regions are represented by a set of n uninitialized logical variables and the colors by a set of m constant values (e.g., {`"red","blue"`}*). The map is modeled by an undirected graph and it is represented as a set whose elements are sets containing two neighboring regions. At the end of the computation each* `Lvar` *representing a region will be initialized with one of the given color.*

```
public static void coloring(Set regions, Set map, Set colors)
throws Failure
{
    Lvar x = new Lvar();
    Solver.add(regions.eq(colors));
    Solver.forall(x,colors,(Set.empty.ins(x)).nin(map));
    Solver.solve();
    return;
}
```

The solution uses a pure "generate & test" approach. The `regions = colors` *constraint allows to find a valuable assignment of colors to regions. Invocation of the* `forall` *method allows to test whether the constraint* {`x`} $\notin$ `map` *holds for all* `x` *belonging to* `colors`. *If it holds, it means that for no pair* $\{r_i, r_j\}$ *in* `map`, r_i *and* r_j *have got the same color.*

If `coloring` *is called with* `regions` = {`r1,r2,r3`}, `r1`, `r2`, `r3` *uninitialized logical variables,* `map` = {{`r1,r2`},{`r2,r3`}}, *and* `colors` = {`"red","blue"`}, *the invocation terminates with success, and* `r1`, `r2`, `r3` *are initialized to* `"red"`,

"blue", *and* "red", *respectively (actually, also the other solution which initializes* r1, r2, r3 *to* "blue", "red", *and* "blue", *respectively, can be computed through backtracking, if the first computed solution turns out to cause a failure).*

Note that the set of colors can be also partially specified. For example, if colors = {c1,"blue"}, *with* c1 *an uninitialized variable, executing* coloring *will generate the constraint:* r1 = Lvar_1, r2 = "blue", r3 = Lvar_1, Lvar_1 ≠ "blue".

5 Defining new constraints

Nondeterminism in JSetL is confined to constraint solving. One consequence of this is that the value of a logical variable computed in a nondeterministic way (hence, within the constraint solver), is no longer "sensible" to backtracking once it is used outside constraint solving. For example, let us consider the following program fragment, where we assume that s is the set {0,1}, and c_1, c_2 are two constraints:

```
Solver.add(x.in(s));
Solver.solve();
if (x == 0) Solver.add(c1);
else Solver.add(c2);
```

If, when evaluating the **if** condition, the value of the logical variable x is 0 then the constraint c_1 is added to the constraint store. If, subsequently, a failure is detected, backtracking will allow to consider a different value for x, namely 1, but the **if** condition is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for x but still with constraint c_1 added to it.

The problem is caused by the fact we cannot guarantee a tight integration between the constraint solver (which is defined in a library) and the primitive constructs of the language. This is probably the main difference between what we called the "library" approach and the approach based on the definition of a new language (or the extension of an existing one). As a matter of fact the problem illustrated by the above program fragment is easily programmed in a language such as SINGLETON where nondeterminism and logic variables are embedded in the language.

However, JSetL provides a solution to overcome this difficulty. The solution is based on the possibility to introduce user-defined new constraints. Whenever a method which the user wants to define requires some nondeterministic action embedded in a non-trivial control structure, one can define the method as a new constraint, so that its execution is completely performed in the context of constraint solving. JSetL provides a class, called NewConstraints, which is devoted to contain all definitions of the new constraints possibly introduced by the user (actually this task would be strongly simplified by the use of a suitable preprocessor that allows most of the details involved in the definition of a new constraint to be hidden to the user).

Let us see how the user can define a new constraint using a simple example: a fully nondeterministic recursive definition of the classical list concatenation operation (`concat`). The solution can be easily generalized to other cases. First of all, the user has to add the following method to the class `NewConstraints`:

```
public static StoreElem concat(Lst l1, Lst l2, Lst l3)
    {
    StoreElem s = new StoreElem(n,l1,l2,l3);
    return s;
    }
```

where n is an integer selected by the user and greater than 100, that will be used by the solver to identify the new constraint. This method returns an instance of `StoreElem`, that is a constraint: hence, it associates the method `concat` with a new constraint, internally identified by the number n. Then the user has to add a new `case` block to the `user_code` method of class `NewConstraints` as follows:

```
protected static void user_code(int c, StoreElem s)
    { ...
    switch(c)
        { ...
        case n: concat(s); break;
        ...}
    ...}
```

Finally it is necessary to define a method `concat` that takes as its input the store element `s` and implements the actual constraint handling procedure for the new constraint. To exploit nondeterminism within this method, one has to adhere to some programming conventions. Let the definition of the new method to be based in general on k different nondeterministic alternatives. Then the user must provide a **switch** statement with k case blocks (numbered from 0 to $k-1$), one for each nondeterministic alternative as follows:

```
public static void concat(StoreElem s) throws Failure
    {
    Lst l1 = (Lst)s.arg1;
    Lst l2 = (Lst)s.arg2;
    Lst l3 =(Lst)s.arg3;
    switch(s.caseControl)
        {
        case 0:
            add_ChoicePoint(s);
            add(l1.eq(Lst.empty));
            add(l2.eq(l3));
            return;
        case 1:
            Lvar x = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
```

```
            add(l1.eq(l1new.ins1(x))); // l1 = [x | l1new]
            add(l3.eq(l3new.ins1(x))); // l3 = [x | l3new]
            add(concat(l1new,l2,l3new)); // concat(l1new,l2,l3new)
            return;
        }
    }
```

The control expression of the **switch** statement is the `caseControl` attribute of the constraint `s` associated with `concat` (default value: 0). Each `case` block, but the last one, creates a choice point and adds it to the stack of the alternatives by executing the statement `add_ChoicePoint(s)`; then the remaining code of the `case` block adds the constraints necessary to compute one of the possible solutions.

Execution of the statement

```
Solver.add(NewConstraint.concat(l1,l2,l3)
```

causes the user-defined constraint `concat` to be added to the current constraint store. If `l1` is `[1,2,3]`, `l2` is `[4,5]`, `l3` is an uninitialized list, a subsequent call to `Solver.solve()` will set `l3` equal to `[1,2,3,4,5]`.

Note that `concat` can be used both to check if a given concatenation of lists holds and to build any of the three lists, starting from any of the other two (like in the usual well-known definition of the `append` predicate in Prolog).

6 Conclusions and future work

We have presented the main features of the JSetL library and we have shown how they can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs. In particular we have described the (set) constraint handling facilities supported by our library and we have shown how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. Furthermore we have shown how to exploit nondeterminism, possibly by introducing new used-defined constraints.

JSetL is fully implemented in Java and can be obtained—as a `.jar` file (67KB)—from the authors.

As a future work the constraint solving capabilities of JSetL could be strongly enhanced by enlarging the constraint domain from that of sets to that of *finite domains*. Following [6], this enhancement could be obtained by integrating an existing constraint solver for finite domains, possibly written in Java, with the JSetL constraint solver over sets. As shown in [6] this would allow us to have, in many cases, the efficiency of the finite domain solvers, while maintaining the expressive power and flexibility of the set constraint solvers (which in turn is inherited from CLP($\mathcal{SET}$)).

On a different side, another concrete improvement could be obtained by using flexible preprocessing tools for the Java language that would allow us to develop suitable syntax extensions that would make it simpler and more natural using the JSetL facilities.

Acknowledgments

The work is partially supported by MIUR project: *Automatic Aggregate—and number—Reasoning for Computing.*

References

[1] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.

[2] K.R. Apt and A. Schaerf. Programming in Alma-0, or Imperative and Declarative Programming Reconciled. In *Frontiers of Combining Systems 2*, D. M. Gabbay and M. de Rijke (editors), Research Studies Press Ltd, 1-16, 2000.

[3] P.Codognet and D.Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3), 185-226, 1996.

[4] A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[5] A.Colmerauer. An introduction to Prolog III. *C-ACM*, 33(7), 69-90, 1990.

[6] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *12th Int'l Workshop on Functional and (constraint) Logic Programming*, Valencia, June 2003.

[7] D.Diaz and P.Codognet. A minimal extension of the Wam for CLP(FD). In *Proc. of the 10th Int'l Conference on Logic Programming*, 1993.

[8] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[9] A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at `http://www.cs.nmsu.edu/TechReports`).

[10] M.Dincbas, P.Van Hentenryck, H.Simonis, et al. The constraint logic programming CHIP. In *Proc. of the 2nd Int'l Conf. On Fifth Generation Computer Systems*, 683-702, 1988.

[11] ECLiPSe, User Manual. Tech. Rept., Imperial College, London. August 1999. Available at `http://www.icparc.ic.ac.uk/eclipse`.

[12] E.Hyyonen, S.DePascale, and A.Lehtola. Interval constraint satisfaction tool INC++. In *Proc. of the 5th ICTAI*, IEEE Press, 1993.

[13] ILOG Optimisation Suite - White Paper. Available at `http://www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.

[14] J.Jaffar, S.Michaylov, P.J.Stuckey, and R.H.C.Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS 14(3)*, 339-395, 1992.

[15] Jean-Francois Puget, and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.

[16] F.Benhamou et al. Le manuel de Prolog IV, PrologIA, June 1996.

[17] G.Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Electronic Notes in Theoretical Computer Science, Vol. 76, Elsevier Science B. V., 17 pages, 2002.

[18] I.Shvetsov, V.Telerman, and D.Ushakov. NeMo+: Object-oriented constraint programming environment based on subdefinite models. In *Artificial Intelligence and Symbolic Mathematical Computations* (G.Smolka, ed.), LNCS 1330, Springer-Verlag, 534-548.

[19] P.Van Hentenryck, H.Simonis, and M.Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence 58*, 113-159, 1992.

[20] Neng-Fa Zhou. DJ: Declarative Java, Version 0.5, User's manual. Kyushu Institute of Tecnology, 1999. Available at `http://www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm`.

[21] Neng-Fa Zhou. Building Java Applets by using DJ - a Java Based Constraint Language. Available at `http://www.sci.brooklyn.cuny.edu/ zhou`.

SML2Java: A Source to Source Translator

Justin Koser, Haakon Larsen, and Jeffrey A. Vaughan

Cornell University

Abstract. Java code is unsafe in several respects. Explicit null references and object downcasting can cause unexpected runtime errors. Java also lacks powerful language features such as pattern matching and first-class functions. However, due to its widespread use, cross-platform compatibility, and comprehensive library support, Java remains a popular language.
This paper discusses SML2Java, a source to source translator. SML2Java operates on type checked SML code and, to the greatest extent possible, produces functionally equivalent Java source. SML2Java allows programmers to combine existing SML code and Java applications.
While direct translation of SML primitive types to Java primitive types is not possible, the Java class system provides a powerful framework for emulating SML value semantics. Function translations are based on a substantial similarity between Java's first-class objects and SML's first-class functions.

1 Introduction

SML2Java is a source-to-source translator from Standard ML (SML), a statically typed functional language [8], to Java, an object-oriented imperative language. A successful translator must emulate distinctive features of one language in the other. For instance, SML's first-class functions are mapped to Java's first-class objects, and an SML let expression could conceivably be translated to a Java interface containing an 'in' function, where every let expression in SML would produce an anonymous instantiation of the let interface in Java. Similarly, many other functional features of SML are translated to take advantage of Java's object-oriented style. Because functional features such as higher-order functions must ultimately be implemented using first-class constructs, we believe one can only achieve a clean design by taking advantage of the strengths of the target language.

SML2Java was inspired by problems encountered teaching functional programming to students familiar with the imperative, object-oriented paradigm. It was developed for possible use as a teaching tool for Cornell's CS 312, a course in functional programming and data structures. For the translator to be a successful educational tool, the translated code must be intuitive for a student with Java experience.

On a broader level, we wish to show how functional concepts can be mapped to object-oriented imperative concepts through a thorough understanding of each

model. In this regard, it becomes important not to force functional concepts upon an imperative language, but rather to translate these functional concepts to their imperative equivalents.

2 Translation

This section discusses the choices we made in our translation of SML to Java. Where pertinent, we will also discuss the benefits and drawbacks of our design decisions.

2.1 Primitives

SML primitive types, such as *int* and *string*, are translated to Java classes. The foregoing become *Integer2* and *String2*, respectively. Ideally, SML primitives would translate to their built-in Java equivalents (e.g. *int* → *java.lang.Integer*), but these classes (e.g. *java.lang.Integer*) do not support operations such as integer addition or string concatenation [10]. We do not map directly to *int* and *string* because Java primitives are not derivatives of *Object*, cannot be used with standard Java collections, and are not compatible with our function and record translations. The latter will be shown in sections 2.2 and 2.4. Our classes, which are based on *Java.util.**, include necessary basic operators and fit well with function and record translation. While Hicks [6] addresses differences between the SML and Java type systems, he does not discuss interoperability. Blume [4] treats the related problem of translating C types to SML.

Figure 1 demonstrates a simple translation. The astute reader will notice several superfluous typecasts. Some translated expressions formally return *Object*. However, because SML code is typesafe, we can safely downcast the results of these expressions. The current version of SML2Java is overly conservative, and inserts some unnecessary casts. Additionaly, the add function looks quite complicated. This is consistent with other function translations which are discussed in section 2.4.

2.2 Tuples and Records

We follow the SML/NJ compiler (section 3) which compiles tuples down to records. Thus, every tuple of the form *(exp1, exp2, ...)* becomes a record of the form {*1=exp1, 2=exp2, ...*}. This should not surprise the avid SML fan as SML/NJ will, at the top level, represent the record {*1="hi", 2="bye"*} and the tuple *("hi","bye"): string*string* identically.

The *Record* class represents SML records. Every SML record value maps to an instance of this class in the Java code. The *Record* class contains a private data member, *myMapping*, of type *java.util.HashMap*. SML records are translated to a mapping from fields (which are of type *String*) to the data that they carry (of type *Object*). The *Record* class also contains a function *add*, which

Fig. 1. Simple variable binding

SML Code:

```
1  val x=40
2  val y=2
3  val z=x+y
```

Java Equivalent:

```
1  public class TopLevel {
2    public static final Integer2 x = (Integer2)
3      (new Integer2 (40));
4
5    public static final Integer2 y = (Integer2)
6      (new Integer2 (2));
7
8    public static final Integer2 z = (Integer2)
9      (Integer2.add()).apply(((
10     (new Record())
11       .add("1", (x)))
12       .add("2", (y))));
13
14 }
```

takes a *String* and an *Object* as its parameters and adds these to the mapping. A record of length n will therefore require n calls to *add*. Record projection is little more than a lookup in the record's *HashMap*.

Fig. 2. Two records instantiated

SML Code:

```
1  val a = {name="John␣Doe", age=20}
2  val b = ("John␣Doe", 20)
```

Java Equivalent:

```
1  public static final Record a = (Record)
2    ((new Record())
3      .add("name", new String2("John␣Doe")))
4      .add("age", (new Integer2 (20)));
5
6   public static final Record b = (Record)
7    ((new Record())
8      .add("1", new String2("John␣Doe")))
9      .add("2", (new Integer2 (20)));
```

2.3 Datatypes

An SML datatype declaration creates a new type with one or more constructors. Each constructor may be treated as a function of zero or one arguments. SML2Java treats this model literally. An SML datatype, *dt*, with constructors *c1, c2, c3 ...* is translated to a class. This class, also named *dt*, has static methods *c1, c2, c3 ....* Each such method returns a new instance of *dt*.

Thus, SML code invoking a constructor becomes a static method call in the translated code. It is important to note that this process is different from the translation of normal SML functions. The special handling of type constructors greatly enhances translated code readability.

A datatype translation is given in figure 3. As the SML langauge enforces type safety, constructor arguments can simply be type *Object*. Although more restrictive types could be specified, there is little benefit in the common case where the type is *Record*.

Fig. 3. A dataype declaration and instantiation

SML Code:

```
1  datatype qux = FOO | BAR of int
2
3  val myVariable = FOO
4  val myOtherVar = BAR(42)
```

Java Equivalent:

```
1  public class TopLevel {
2    public static class qux extends Datatype {
3
4      protected qux(String constructor){
5        super(constructor);
6      }
7
8      protected qux(String constructor,Object data){
9        super(constructor, data);
10     }
11
12     public static qux BAR(Object o){
13       return new qux("BAR", o);
14     }
15
16     public static qux FOO(){
17       return new qux("FOO");
18     }
19
20   }
21   public static final qux myVariable = (qux)
22     qux.FOO();
23
24   public static final qux myOtherVar = (qux)
25     qux.BAR((new Integer2 (42)));
26
27 }
```

2.4 Functions

In our translation model, the *Function* class encapsulates the concept of an SML function. Every SML function becomes an instance of this class. The Java *Function* class has a single method, *apply*, which takes an *Object* as its only parameter and returns an *Object*. The *Function* class encapsulation is necessitated by the fact that functions are treated as values in SML. As a byproduct of this scheme, function applications become intuitive; any application is translated to *Function_Name.apply(argument)*.

At an early design stage, the authors considered translating each function to a named class and a single instantiation of that class. While this model provides named functions that can be passed to other functions and otherwise treated as data, it does not easily accommodate anonymous functions. A strong argument for the current model is that instantiating anonymous subclasses of *Function* provides a natural way to deal with anonymous functions.

We believe this is a sufficiently general approach, and can handle all issues with respect to SML functions (including higher-order functions). In fact, every SML function declaration (i.e. named function) is translatead, by the SML/NJ compiler, to a recursive variable binding with an anonymous function. Therefore our treatment of anonymous functions and named functions mirror each other and this similarity lends itself to code readability.

Other authors have used different techinques for creating functions at runtime. For example, Kirby [7] uses the Java compiler to generate bytecode dynamically. While powerful and well suited for imperative programming, this approach is not compatible with the functional philosophy of SML.

In figure 4, the lines that contain the word "Pattern" form the foundation of what will, in future revisions of SML2Java, be fully generalized pattern matching. Pattern matching is done entirely at runtime, and consists of recursively comparing components of an expression's value with a pattern. SML/NJ performs some optimizations of patterns at compile time [1]. However these optimizations are, in general, NP-hard [3] and SML2Java does not support them. Currently patterns are limited to records (including tuples), wildcards and integer constants.

2.5 Let Expressions

A *Let* interface in Java encapsulates the SML concept of a let expression. The *Let* interface has no member functions. Every SML let expression becomes an anonymous instantiation of the *Let* interface with one member function, *in*. This function has no parameters and returns whatever type is appropriate given the original SML expression. The *in* function is called immediately following object instantiation.

A different approach would be to have the *Let* interface contain the function *in*. Here, *in* would have no formal parameters, and would return an *Object*. The advantage to this would be its consistency with respect to our function

Fig. 4. Named function declaration and application

SML Code:

```
1  val getFirst = fn(x:int, y:int) => x
2  val one = getFirst(1,2)
```

Java Equivalent:

```
1  public static final Function getFirst = (Function)
2    (new Function () {
3      Object apply(final Object arg) {
4        final Record rec = (Record) arg;
5        RecordPattern pat = new RecordPattern();
6        pat.add("1", new VariablePattern(new Integer2()));
7        pat.add("2", new VariablePattern(new Integer2()));
8        pat.match(rec);
9        final Integer2 x = (Integer2) pat.get("1");
10       final Integer2 y = (Integer2) pat.get("2");
11       return (Integer2) (x);
12     }
13   });
14
15 public static final Integer2 one = (Integer2)
16   (getFirst).apply(((
17   (new Record())
18     .add("1", (new Integer2 (1))))
19     .add("2", (new Integer2 (2)))));
```

translations (i.e. the *apply* function), but a possible disadvantage is excessive typecasting, which can greatly reduce readability.

One might also attempt to separate the *Let* declaration from the call to its *in* function. If implemented in the most direct manner, such a model would, like the previous one, require that the *Let* interface contain an *in* function. This scheme would improve code readability. However, as one often has several *Let* expressions in the same name-space in SML, this model would likely suffer from shadowing issues.

Fig. 5. Let expressions are translated like functions

SML Code:

```
1  val x =
2    let
3      val y = 1
4      val z = 2
5    in
6      y+z
7    end
```

Java Equivalent:

```
1  public static final Integer2 x = (Integer2)
2    (new Let() {
3      Integer2 in() {
4        final Integer2 y = (Integer2) (new Integer2 (1));
5        final Integer2 z = (Integer2) (new Integer2 (2));
6        return (Integer2) (Integer2.add()).apply(((
7        (new Record())
8          .add("1", (y)))
9          .add("2", (z))));
10     }
11   }).in();
```

2.6 Module System

Our translation of SML's module system is straightforward. SML signatures are translated to abstract classes. SML structures are translated to classes that extend these abstract signature classes. A structure class only extends a given signature class if the original SML structure implements the SML signature. Structure declarations that are not externally visible in SML (i.e. not included

in the implemented signature) are made private data-members in the generated Java structure class. This is demonstrated in figure 6.

3 Implementation

Our primary task was to translate high-level SML source code to high-level Java source code. As there are several available implementations of SML, we chose to use the front end of one, Standard ML of New Jersey (SML/NJ) [9]. We use the development flavor of the compiler (`sml-full-cm`) to parse and type-check input SML code. We then translate the abstract syntax tree generated by SML/NJ to our own internal Java syntax tree and output the Java code in source form.

Taking advantage of the SML/NJ type checker gives us a strong guarantee regarding the safety of the code we are translating. To cite Dr. Andrew Appel, a program produced from this code "cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics" [2]. In other words, such a program cannot dump core, access private fields, or mistake types for one another. It would be interesting to investigate whether these facts, combined with the translation semantics of SML2Java, imply that similar guarantees hold in the generated Java code.

Other properties of the Core subset of SML are discussed by VanIngwegen [12]. Using HOL [5], she is able to prove, among other things, determinism of evaluation.

4 Conclusion and Future Goals

The current version of SML2Java translates many core constructs of SML, including primitive values, datatypes, anonymous and recursive functions, signatures and structures. SML2Java succeeds in translating SML to Java code, while respecting the functional paradigm.

Parametric polymorphism is a key construct that the authors would like to implement in SML2Java. Java 1.5 (due out late 2003) will directly support generics [11], and we believe waiting for Sun's implementation will facilate generating clean Java code. In addition, Java's generics will resemble C++ templates, and our treatment of parametric polymorphism should highlight the relative merits of each approach.

We would like to add support for several less critical SML constructs. Among these are exceptions, vectors, open declarations, mutual recursion, functors, and projects containing multiple files. The majority of these should be implementable without excessive difficulty, and each is expected to be a valuable addition to SML2Java.

Fig. 6. Translation of a signature and a structure

SML Code:

```
signature INDEX_CARD = sig
  val name : string
  val age : int
end

structure IndexCard :> INDEX_CARD = struct
  val name = "Professor␣Michael␣Jordan"
  val age = 31
  val super_secret = "This␣secret␣cannot␣be␣visible␣to␣the␣
    outside"
end
```

Java Equivalent:

```
public class TopLevel {
  private static abstract class INDEX_CARD  {
    public static final String2 name = null;
    public static final Integer2 age = null;
  }

  public static class IndexCard extends INDEX_CARD {
    public static final String2 name = (String2)
      (new String2 ("Professor␣Michael␣Jordan"));

    public static final Integer2 age = (Integer2)
      (new Integer2 (31));

    private static final String2 super_secret = (String2)
      (new String2 ("This␣secret␣cannot␣be␣visible␣to␣the␣
        outside"));

  }

}
```

5 Acknowledgements

This project was performed as independant research under the guidance of Dexter Kozen, Cornell University. We would like to thank Professor Kozen for many insightful discussions and much valuable advice. We would also like to thank the following for helpful advice: Andrew Myers, Cornell University, and Tore Larsen, Tromsø University.

References

[1] Aitken, William. *SML/NJ Match Compiler Notes* http://www.smlnj.org/compiler-notes/matchcomp.ps (1992)

[2] Appel, Andrew W. *A critique of Standard ML* http://ncstrl.cs.princeton.edu/expand.php?id=TR-364-92 (1992)

[3] Baudinet, Marianne and MacQueen, David. *Tree Pattern Matching for ML (extended abstract)* http://www.smlnj.org/compiler-notes/85-note-baudinet.ps (1985)

[4] Blume, Matthias. *No-Longer-Foreign: Teaching an ML compiler to speak C "natively"* Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)

[5] Gordon, Melham *Introduction to HOL. A theroem proving environment for higher order logic* Cambridge University Press, 1993

[6] Hicks, Michael. *Types and Intermdiate Representations* University of Pennsylvania (1998).

[7] Kirby, Graham, et al. *Linguistic Reflection in Java* Software - Practice & Experience 28, 10 (1998).

[8] Milner, Robin, et al. *The Definition of Standard ML - Revised.* Cumberland, RI: MIT Press, 1997.

[9] SML/NJ Fellowship, The. *Standard ML of New Jersey* http://www.smlnj.org (July 29, 2003).

[10] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2 API Specification* http://java.sun.com/j2se/1.4.2/docs/api/ (July 18, 2003).

[11] Sun Microsystems. *JSR 14 Add Generic Types To The Java Programming Language* http://www.jcp.org/en/jsr/detail?id=14 (July 24, 2003).

[12] VanIngwegen, Myra. *Towards Type Preservation for Core SML* http://www.myra-simon.com/myra/papers/JAR.ps.gz

Constraint Imperative Programming with C++

Olaf Krzikalla

Reico GmbH krzikalla@gmx.de

Abstract. Constraint-based programming is of declarative nature. Problem solutions are obtained by specifying their desired properties, whereas in imperative programs the steps that lead to a solution must be defined explicitly. This paper introduces the Turtle Library, which combines constraint-based and imperative paradigms. The Turtle Library is based on the language Turtle[1] and enables constraint imperative programming with C++.

1 Constraint Imperative Programming at a Glance

In an imperative programming language the programmer describes how a solution for a given problem has to be computed. In contrast to that, in a declarative language the programmer specifies what has to be evaluated. Constraint-based programming is a rather new member of the declarative paradigm that was first developed from logic programming languages. In constraint-based programming the programmer describes the solution only by specifying the variables, their properties and the constraints over the set of variables. Actually, no algorithms have to be written. The compiler and run-time environment are responsible for providing appropriate algorithms and eventually obtaining a solution.

Meanwhile, constraint-based programming has been extended by concepts of other - mostly declarative - programming languages. However, the combination of imperative and constraint-based languages is far less explored. Borning and Freeman-Benson[2] introduced the term 'constraint-imperative programming' and developed the language Kaleidoscope[3], combining constraint and object-oriented programming. But object-orientation is no precondition for constraint-imperative programming. This paper deals with more fundamental problems of the integration of constraints and constraint solvers in imperative language concepts. This integration promises some advantages. Imperative programming is a well known paradigm, which is intuitively understood by most programmers. A lot of efficient and industrial-strength imperative languages exist. However, an imperative program for a difficult algorithm is sometimes very cumbersome. Especially for this sort of problems declarative languages have proven their power. Constraint programming enables the programmer to specify required relations between objects directly rather than to ensure these relations by algorithms only. So constraint programs not only often become more compact and readable, but also less erroneous than their imperative counterparts.

Constraint imperative programming tries to combine the advantages of constraint-based and traditional imperative programming. A recent development in

this field is the language Turtle, a constraint imperative programming language developed by Martin Grabmüller at the Technische Universität Berlin. Based on the ideas presented in [1] I developed the Turtle Library, a constraint imperative programming approach in C++.

2 The Basic Concept of Turtle

The fundamental difference between imperative and declarative languages is the model of time. In pure declarative languages a timing model simply does not exist - computations are specified independent of time. On the other hand, an imperative language always describes transformations of a given state at one point in time to another state at the next point in time. Computations are specified by sequences of statements.

Whenever declarative and imperative languages are combined, one of the main issues is the interaction of the integrated declarative concepts with the imperative timing model. In Turtle this is solved by introducing a lifetime for constraints and the statement *require*, which defines a constraint:

```
require constraint;
```

When a *require* is reached during the execution of the program, the given constraint is added to a global constraint store and taken into account during further computations - its lifetime starts. A constraint doesn't exist (and the system doesn't know anything about it) until the corresponding *require*-statement is executed. Eventually a sequence of *require*-statements form a conjunction of the appropriate constraints in the constraint store. Constraints in the constraint store are considered active.

Of course, if a constraint starts to exist at a certain time, it also can be removed at a certain time:

```
require constraint in
  statement;
  ...
end;
```

The given constraint exists only between the *in* and *end*. When the program reaches the *end* statement (or otherwise leaves the block), the constraint is removed from the constraint store - its lifetime ends. After this the constraint isn't active any longer.

In order to deal with over- and underconstrained problems constraints need to be labelled with strenghts to form a constraint hierarchy. Altough a constraint imperative system without constraint hierarchies could be designed, its usefulness would be drastically reduced, because it would be difficult to constrain variables while the program dynamically adds or removes constraints. In Turtle each constraint can have a strength annotation in its definition:

```
require constraint1 : strong;
require constraint2 : mandatory;
```

When a constraint is annotated with a strength, it is added to the store with the given strength, otherwise with the strongest strength *mandatory*. This strength was specified in the previous example for clarity only.

Constraints are defined on constrainable variables. Most of the time a constrainable variable acts like a normal variable: it can be used in expressions and as a function argument. Only in a constraint statement they differ from their normal counterparts. A normal variable is treated like a constant, but a constrainable variable acts like a variable in the mathematical sense, and the constraint solver may change its value in order to satisfy all constraints existing at this point in time.

```
var x : int; // a normal variable
var y : ! int; // the exclamation defines a constrained variable
x := 0;
require y <= x in
  ... // during the execution of this block Turtle ensures y <= 0
end;
```

Constraints in Turtle are boolean expressions. During the execution of a *require* statement the constraint solver computes a certain value for each constrained variable, such that all active constraints evaluate to true. Constraints are handled strictly *eager*. Changing a non-constrained variable after it was used in a constraint doesn't affect the constraint store. Whenever the program reads a constrained variable, the value last computed by the solver for this variable is supplied. An exception is raised, if it isn't possible to satisfy all mandatory constraints during the execution of a *require* statement.

In Turtle constraints can be used for computing solutions to a certain problem like other constraint programming approaches. But they are not limited to this usage. *require* statements introduce conditions *a priori*, which are maintained automatically by the constraint solver. Hence backtracking like in approaches with *a posteriori* tests (e.g. Alma-0[6]) is not neccessary. Due to the *a priori* nature of constraints in Turtle they can be used to describe and preserve program invariants or - more general - to express in declarative manner the meaning of an otherwise imperative program without disrupting the familiar execution flow.

3 A Turtle in C++

The concepts of Turtle were first implemented in a language developed from scratch. This approach was chosen because some other features like higher-order functions should also be integrated. And a new language seemed to be the best choice for the seamless combination of imperative, functional and constraint programming. However, a new language is always in a difficult position. The

knowledge base is small, tools don't exist, and further development is sometimes driven by academic interests only.

All concepts of Turtle related to constraint programming are also implementable in C++. Thats why I think a Turtle Library written in pure C++ serves both the widespreading and further development of Turtle better. In the recent years a lot of developments - especially on the field of generic programming in C++ - made it possible to move almost all concepts from the Turtle language to the C++ Turtle Library without any losses. Furthermore, the generic approach of the Turtle Library enables every user to add, change or optimize constraint solvers at will. This is especially important for user-defined domains and offers a wide application field for the Turtle Library. The Turtle Library might be used to solve operational research problems or to program a graphical user interface. Both problems are typical constraint problems. In the first problem constraint programming is used only to obtain a solution, which often can be done in a constraint logic language too (given an appropriate language and - more important - an appropriate programmer) or by using a rather imperative approach[5]. But for the second problem constraint imperative programming really shines. The 'canonical' example is a graphical element, which can be dragged by the mouse inside certain borders[4]. The imperative approach looks like this:

```
void drag ()
{
  while (mouse.pressed) {  //message processing is left out
    int y = mouse.y;
    if (y > border.max)
      y = border.max;
    if (y < border.min)
      y = border.min;
    draw_element (fix_x, y, graphic);
  }
}
```

Using the Turtle Library the example would look as follows:

```
void drag ()
{
  constrained<int> y;
  require (y >= border.min && y <= border.max);
  while (mouse.pressed) {
    y = mouse.y;
    draw_element (fix_x, y(), graphic);
  }
}
```

The above is not only shorter, but expresses the relation between the border-object and the y-coordinate in exactly the way a programmer would think about it.

3.1 Constrained Variables

A constrained variable is of the generic type `constrained`. A constrained variable has identity semantics, the copy constructor and standard assignment operator aren't implemented. If they are needed, an appropriate wrapper (e.g. a reference counted pointer) has to be defined. The public interface given here is described in detail in the following sections.

```
template<class T>
class constrained
{
  public:
    constrained (const T& prefer = T());
    constrained<T>& operator= (const T& prefer);
    ~constrained ();
    T operator ()() const throw (overconstrained_error, ...);
    const T& preferred() const;
    void unfix() const;
};
```

The template parameter specifies the value type of the variable. It might be a fundamental type like `int` or `double` or an user-defined class. Domains are formed by non-intersecting sets of value types and for each domain an appropriate constraint solver has to be provided. Thus each value type is unambiguously bound to a constraint solver. However Turtle can be used for hybrid domains, because the interface enables the implementation of a constraint solver responsible for more than one value type.

3.2 Declaring Constraints

Constraints can be declared as straightforward as presented in the section 2:

```
constrained<double> a, b;
double c = 2.0;
require (a >= 0.0);
require (a <= b && a + b <= c);
```

The composition of the boolean expression inside a `require` is done using operator overloading and expression template techniques. Which operators are supported for a certain value type is defined by the domain and the available constraint solver. E.g. it is rather pointless to support `>`, `<` or `!=` for floating point values[1]. In domains other than the algebraic ones it's often better to avoid otherwise meaningless operator overloading. For this purpose named predicates can be defined and used instead:

[1] Due to the same reasons even the support of == could be argued.

```
edge e = /*...*/;  //compute an edge
constrained<vertex> p;
require (point_on_edge (e, p));
```

The operator && forms a conjunction of two expressions just like two subsequent requires, hence

```
constrained<double> a;
require (a >= 0 && a <= 2);
```

is equivalent to

```
constrained<double> a;
require (a >= 0);
require (a <= 2);
```

The operator || defines a disjunction. A disjunction can be seen as a branch in a tree of solutions. Subseqent requires add their constraints to all leafs of the tree.

```
constrained<double> a, b;
require (a == 0 || a == 1);
require (b == a + 1);
// the store now contains :
// (b == a + 1 && a == 0) || (b == a + 1 && a == 1)
```

The Turtle Library provides a simple generic algorithm for handling disjunctions. A certain constraint solver may implement a more sophisticated approach to compute and maintain solution trees efficiently.

Constraint strengths can be given as a second argument to require like in the Turtle language:

```
require (a == b, weak);
```

Of course these values are only of interest if the underlying constraint solver supports hierarchic constraints.

The Turtle Library internally stores the constraints in several constraint substores. A constraint sub-store is defined as the set of all constraints over a set of constrained variables, where each variable of the set is linked to each other variable of the set. Two variables x and y are linked, if they either both appear in one constraint or if x appears in a constraint containing a variable linked to y.

```
constrained<double> a, b;
require (a >= 0.0);  // generate constraint sub-store 1
require (b >= 0.0);  // generate constraint sub-store 2
require (a <= b);    // sub-store 1 and 2 are merged together
```

The function template `require` returns a handle to manage the lifetime of the constraint. If the return value is ignored, the imposed constraint exists as long as all constrained variables in this constraint:

```
constrained<int> a;
{
  constrained<int> b;
  require (a == b);
  //...
//leaving the scope of b, hence a == b
//is removed from the constraint store:
}
```

Otherwise, the lifetime of the constraint is also bound to the lifetime of the returned constraint handle:

```
constrained<int> a, b;
{
  constraint_handle<int> z = require (a == b);
  //...
//leaving the scope of z, hence a == b
//is removed from the constraint store:
}
```

Still, the constraint exists no longer than all constrained variables in it. When the handle ceases to exist after the constraint did, it is ignored.

3.3 Obtaining Values from Constrained Variables

In a first version of the Turtle Lib, the `constrained<T>` class has an `operator T() const` member function to obtain the actual value of the constrained variable. However, it turns out that this operator sometimes conflicts with the generation of expression templates in a `require`. Thus the function call operator `operator()() const` was overloaded to read a value from a constrained variable:

```
std::cout << a();  //prints a value matching all constraints to a
```

Whenever this operator is invoked, the constraint solver is started to determine the value of the appropriate variable. How the value is determined depends mainly on the solver. When the store is overconstrained and no value can be determined, an exception of type `overconstrained_error` (derived from `std::logic_error`) is raised.

But more often underconstrained situations occur. For this purpose the Turtle Library supports a preferred value. A value of type T can be assigned to a `constrained<T>` or used to construct such a variable. This value then becomes the preferred value of the constrained variable. Now, if it turns out that more than one solution exists for a certain variable, the solution closest to the preferred value is taken:

```
constrained<double> a (3);
require (a <= 2.5);
std::cout << a();  // prints 2.5
```

To a certain degree the preferred value acts like a weak constraint. This is especially useful, if the constraint solver itself doesn't support constraint hierarchies. Thus a hierarchic constraint solver isn't as necessary as in the original Turtle language.

The evaluation of the preferred value is done by the solver implementation. It can be used to define a *threshhold* or *destination* value enabling the solver to terminate the search through the solution tree as soon as possible.

Some domains consist of incompareable values making it impossible to define a closest solution. In this case no general behaviour can be defined. Instead the solver implementation has to define the use of the preferred value.

3.4 Implicit Fixing

Once a value is determined for a constrained variable, this value has to be taken into account for further calculations. The constrained variable itself gets implicitly fixed to the determined value:

```
constrained<int> a (2), b (0);
require (a == b);
std::cout << a();  // prints 2
std::cout << b();  // also prints 2
```

Without implicit fixing the value of b would be evaluated to 0 and hence violate the required constraint a == b. Implicit fixing is done by generating a new constraint of the form variable == value. Due to this important side effect the evaluation order of constrained variables must be carefully considered. If the output lines of the above example were exchanged, both lines would print 0. And the following leads to unspecified behavior:

```
std::cout << a() << b();  // which variable is evaluated first?
```

The implicit fix is not immediatly added to the constraint sub-store but kept in a delay store inside the sub-store. If only one implicit fix exists in a constraint sub-store, and the same variable shall be evaluated again, the fix is erased before the evaluation (later in the process a new fix will be added). If more implicit fixes exist, always all are taken into account.

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
```

```
  // prints j, because the only implicit fixed variable is a:
  std::cout << a();
}

constrained<int> a (2), b (0);
require (a == b);
std::cout << b();  // prints 0, fixes b
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  // always prints 0, because b is fixed, but a is evaluated:
  std::cout << a();
}
```

As shown in the last example, sometimes implicit fixes are harmful, especially if more than one variable is evaluated inside a loop. That's why a constrained variable can be unfixed explicitly via the member function `unfix()`:

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  std::cout << a();  // prints j and get fixed
  std::cout << b();  // prints also j and get fixed
  //now more than one fix exist, so all fixes would be considered
  //during further evaluations unless we explicitly
  //unfix the variables:
  a.unfix();
  b.unfix();
}
```

The computation of a value for a constrained variable differs a lot from the original Turtle language. While in the Turtle language the values of constrained variables are already determined during a require statement, the Turtle Library delays the computation until a read-action to a constrained variable occurs. The disadvantage of this approach seems the need of implicit fixing, which isn't part of the Turtle language[2].

On the other hand the delay of the computation offers some advantages. First, only when the computation is delayed until a read-action, the preferred value can be evaluated correctly. Otherwise a change of the preferred value after some requires could be ignored. Second, a solver knows which constrained variable actually is being read, can consider this fact during the computation

[2] Altough there is an ongoing argument about this topic.

and hence doesn't have to evaluate all variables in every case. And third, lazy evaluation becomes possible. Altough also the Turtle Library handle constraints *eager* mostly, it is not limited to this.

3.5 Lazy Evaluation

Lazy evaluation is an often arising issue when declarative and imperative concepts are combined. Shall a subexpression in a require-statement be evaluated immediately or shall the evaluation be delayed until the constraint is actually needed for the evaluation of a constrained variable? Consider the example:

```
int foo();

int example()
{
  int_c a, b
  int i = 1;
  require (a == i);
  require (b >= foo());
  require (a < b);
  i = 2;
  std::cout << a();  // 1 or 2 ?, is foo() called here ?
  std::cout << b();  // or is foo() called only here ?
}
```

As stated earlier the Turtle Library doesn't perform lazy evaluation by default. This decision was made mainly due to lifetime issues. In C++ it's impossible to ensure that an arbitrary object exists until all constraints referring to it are erased. Hence the above example prints 1 for `a` and calls `foo()` during the evaluation of the argument for the second `require`. This has the additional benefit, that possible side effects of functions inside constraints are more predictable. If `foo()` would be lazy evaluated in the example above, it could be called once or twice, depending on the actual implementation of the underlying constraint solver.

Lazy evaluation can be simulated through the lifetime management of constraints. But sometimes it is just better to have some lazy evaluated values. Therefore a simple lazy evaluated value type is provided by the Turtle Library:

```
template<class T>
class lazy_evaluated
{
  public:
    explicit lazy_evaluated (const T& init = T());
    operator T() const;
    operator T&();
};
```

This class mostly acts like a value of type T, but its actual value is garbage collected (the copy constructor and assignment operator of `lazy_evaluated<T>` has identity semantics). Each constraint using a lazy evaluated variable stores a copy of the corresponding `lazy_evaluated<T>` variable. The actual value is preserved unless all references to it are removed. It is only read by the constraint solver when needed during the evaluation of a constrained variable. Side effects may only happen due to the copying of T.

```
int_c a;
lazy_evaluated<int> i = 1;
require (a == i);
i = 2;
std::cout << a();  //reads i at this point and thus prints 2
```

4 Programming with the Turtle Library

The Turtle Library can be downloaded from
`http://home.t-online.de/home/krize6/turtle.htm`.
At this page also some technical issues are discussed in more detail. Especially the steps needed to integrate a new constraint solver in the Turtle Library are described. Furthermore some more sophisticated examples of constraint imperative programming are already provided. They demonstrate the use of some techniques and little patterns to make constraint imperative programming more convenient and flexible.

4.1 User-defined Constraints and Dynamic Expressions

Often the declarative power of expression templates is sufficient to express the constraints in a compact and readable manner. But some constraints are so common that they deserve an own name. Such user-defined constraints can be generated using the function template `build_constraint`, which takes an constraint just like require, but only builds the internal representation of the given expression without adding it to the constraint store.

```
typedef constrained<int> int_c;

constraint_solver<int>::expr domain (const int_c& x, int min, int max)
{
  return build_constraint (x >= min && x <= max);
}

int_c a, b, c;
require (domain (a, 0, 9));
require (domain (b, 0, 99));
require (domain (c, -1, 1));
```

The naming of complex static expressions further enhances the readability of a program. But besides this constraint imperative programming also needs a way to create constraints dynamically. For this the Turtle Library provides a generic class `dynamic_expr`, which holds an (sub)expression and can be used like that, but has value semantics. A rather complex example is the function `example_dynamic_puzzle`, which is part of the sample file provided on the internet page of the Turtle Library.

4.2 Optimization

Constraint programming supplies a lot of tools to optimize a given function for a given set of constraints. Optimization is one the main usages of constraint programming. Hence, optimization should be possible with the Turtle Library, too. By using a preferred value for a given expression, optimization can be done without the needs of special library functions. Consider the following example:

```
double_c x, y;
require (y >= 0);
require (y >= 3 - 2 * x);
```

Given these constraints the sum of x and y shall be minimized. These can be done by a little pattern of the following three lines:

```
double_c min (- 1000.0);
require (min == x + y);
std::cout << min(); // prints 1.5
```

First a constrained variable has to be declared and the preferred value have to be set to an absolute minimal or maximal border [3]. Second, this variable has to be set equal to the expression to be optimized. And third, by reading the variable the value closest to the given preferred value gets calculated and stored in the variable. Furthermore the implicit fixing also immediately limits other constrained variables to values at the searched optimum.

5 Conclusion and Future Works

The Turtle Library defines an interface for the integration of constraint programming concepts in an imperative language and provides an implementation of this interface for a popular language. Hopes are, that this opens a wider application field for constraint imperative programming. Only the practical use will show further needs. E.g. if an implicit fix of a constrained variable has to be considered is defined by a rather complex rule. It's unclear if this rule is of any practical

[3] This example is rather abstract and hence knows no 'absolute' minimum. In practical applications it should be always possible to find a reasonable value (see also `example_knapsack`).

value. Also, for the moment there is no way to unfix a bunch of variables at once (e.g. all variables of a sub-store).

The class `lazy_evaluated<T>` should be treated as a simple example for lazy evaluation. It is possible to further parametrize this class to allow more complex actions during constraint evaluation including the call of functions. If this is done, side effects of a lazy evaluated function has to be considered carefully as stated in section 3.5. At the moment it's quite unclear if the gain of flexibility outweighs the possiblity of near unpredictable side effects.

The modelling of algebraic problems using the Turtle Library is already very convenient. But the generic approach offers a lot more. A lot of publications in the recent decade has shown, that constraint programming is well-suited for several problem domains. But unfortunately a lot of these publications either introduced a whole new language or at least extended an existing language by adding new language constructs (and thus became incompatible to the parent language). But an application programmer can't just move from one language to the next at will. Due to business, management and also educational issues he has to stick to one - often for years. With the Turtle Library now even the application programmer gets a tool to use constraints in C++ in the convenient declarative manner as it is already used for years in other languages.

References

[1] Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin 2003,

[2] Freeman-Benson, B.N.: Constraint Imperative Programming. PhD Thesis, University of Washington, 1991. Published as Department of Computer Science and Engenieering Technical Report 91-07-02

[3] Borning, A. and Freeman-Benson, B.N.: The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 174-180, 1992

[4] Lopez, G.: The design and implementation of Kaleidoscope, a constraint imperative programming language. PhD Thesis, University of Washington, 1997.

[5] ILOG. ILog Web Site.
`http://www.ilog.com`, last visited 2003-06-23

[6] Apt, K.R., Brunekreef, J., Partington, V. and Schaerf, A.: Alma-0: An imperative language that supports declarative programming. ACM Toplas, 20(5):1014-1066, 1998.

Patterns in Datatype-Generic Programming

Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
jeremy.gibbons@comlab.ox.ac.uk

Abstract. *Generic programming* consists of increasing the expressiveness of programs by allowing a wider variety of kinds of parameter than is usual. The most popular instance of this scheme is the C++ Standard Template Library. *Datatype-generic programming* is another instance, in which the parameters take the form of datatypes. We argue that datatype-generic programming is sufficient to express essentially all the genericity found in the *Standard Template Library*, and to capture the abstractions motivating many *design patterns*. Moreover, datatype-generic programming is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, and thereby offers the prospect of better static checking and a greater ability to reason about generic programs. This paper describes work in progress.

1 Introduction

Generic programming [28, 19] is a matter of making programs more adaptable by making them more general. In particular, it consists of allowing a wider variety of entities as parameters than is available in more traditional programming languages.

The most popular instantiation of generic programming today is through the C++ Standard Template Library (STL). The STL is basically a collection of container classes and generic algorithms operating over those classes. The STL is, as the name suggests, implemented in terms of C++'s template mechanism, and thereby lies both its flexibility and its intractability.

Datatype-generic programming (DGP) is another instantiation of the idea of generic programming. DGP allows programs to be parameterized by a *datatype* or *type functor*. DGP stands and builds on the formal foundations of category theory and the *Algebra of Programming* movement [8, 7, 10], and the language technology of Generic Haskell [22, 12].

In this paper, we argue that DGP is sufficient to express essentially all the genericity found in the STL. In particular, we claim that various programming idioms that can at present only be expressed informally as *design patterns* [17] could be captured formally as datatype-generic programs. Moreover, because DGP is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism

in particular, this observation offers the prospect of better static checking of and a greater ability to reason about generic programs than is possible with other approaches.

This paper describes work in progress — in fact, it describes work largely in the future. The United Kingdom's Engineering and Physical Sciences Research Council is funding a project called *Datatype Generic Programming*, starting around September 2003. The work described in this paper will constitute about a third of that project; a second strand, coordinated by Roland Backhouse at Nottingham, is looking at more of the underlying theory, including logical relations for modular specifications, higher-order naturality properties, and termination through well-foundedness; the remainder of the project consists of an integrative case study.

The rest of this paper is structured as follows. Section 2 describes the principles underlying the C++ Standard Template Library. Section 3 motivates and defines Datatype-Generic Programming, and explains how it differs from a number of similar approaches to genericity. Section 4 discusses the Design Patterns movement, and presents our case for the superiority of datatype genericity over informal prose for capturing patterns. Section 5 concludes by outlining our future plans for the DGP project.

2 Principles Underlying the STL

The STL [6] is structured around four underlying notions: *container types*, *iterators*, *algorithms*, and *function objects*. These notions are grouped into a hierarchy (in fact, a directed acyclic graph) of *concepts*, representing different abstractions and their relationships. The library is implemented using the C++ *template mechanism*, which is the only means of writing generic programs in C++. This section briefly analyzes these six principles, from a functional programmer's point of view.

2.1 The C++ Template Mechanism

The C++ template mechanism provides a means for classes and functions to be parametrized by types and (integral, enumerated or pointer) values. This allows the programmer to express certain kinds of abstraction that otherwise would not be available. A typical example of a function parametrized by a type is the function *swap* below:

$$\begin{array}{l}\textbf{template}\langle\textbf{class } T\rangle\\ \textbf{void } \mathit{swap}(T\&\ a, T\&\ b)\ \{T\ c = a; a = b; b = c;\}\\ \\ \mathit{main}()\ \{\\ \quad \textbf{int } i_1 = 3, i_2 = 4; \qquad\qquad \mathit{swap}\langle\textbf{int}\rangle(i_1, i_2);\\ \quad \textbf{double } d_1 = 3.5, d_2 = 4.5; \ \mathit{swap}\langle\textbf{double}\rangle(d_1, d_2);\\ \}\end{array}$$

The same function template is instantiated at two different types to yield two different functions. Container classes form typical examples of parametrization of a class by a type; the example below shows the outline of a *Vector* class parametrized by size and by element type.

```
template⟨class T, int size⟩
class Vector {private : T values[size]; ...};

main() {
   Vector⟨int, 3⟩ v;
   Vector⟨ Vector⟨double, 100⟩, 100⟩ matrix;
}
```

The same class template is instantiated three times, to yield a one-dimensional vector of three integers and a two-dimensional 100-by-100 matrix of doubles.

A template is to all intents and purposes a macro; little is or can be done with it until the parameters are instantiated, but the instantiations that this yields are normal code and can be checked, compiled and optimized in the usual way. In fact, the decision about which template instantiations are necessary can only be made when the complete program is available, namely at link time, and typically the linker has to call the compiler to generate the necessary instantiations.

The C++ template mechanism is really a *special-purpose, meta-programming* technique, rather than a general-purpose generic-programming technique. Meta-programming consists of writing programs in one language that generate or otherwise manipulate programs written in another language. The C++ template mechanism is a matter of meta-programming rather than programming because templated code is not actually 'real code' at all: it cannot be type-checked, compiled, or otherwise manipulated until the template parameter is instantiated. Some errors in templated code, such as syntax errors, can be caught before instantiation, but they are in the minority; static checking of templates is essentially impossible. Thus, a class template is not a formal construct with its own semantics — it is one of the ingredients from which such a formal entity can be constructed, but until the remaining ingredients are provided it is merely a textual macro. In a programming language that offers such a template mechanism as its only support for generic programming, there is no hope for a calculus of generic programs: at best there can be a calculus of their specific instances.

The template mechanism is a special-purpose, as opposed to general-purpose, meta-programming technique, because only limited kinds of compile-time computation can be performed. Actually, the mechanism provides surprising expressive power: Unruh [38] demonstrated the disquieting possibility of a program whose compilation yields the prime numbers as error messages, Czarnecki and Eisenecker [13] show the Turing-completeness of the template mechanism by implementing a rudimentary LISP interpreter as a template meta-program, and Alexandrescu [4] presents a tour-de-force of unexpected applications of templates. But even if technically template meta-programming has great expressiveness, it is pragmatically not a convenient tool for generating programs; applications of the technique feel like tricks rather than general principles. Ev-

erything computable is expressible, albeit sometimes in unnatural ways. A true general-purpose meta-programming language would support 'programs as data' as first-class citizens, and simple and obvious (as opposed to 'surprising') techniques for manipulating such programs [35].

There are several consequences of the fact that templated code is a meta-program rather than (a fragment of) a pure program. They all boil down to the fact that separate compilation of the templated code is essentially impossible; it isn't real code until it is instantiated. Therefore:

- templated code must be distributed in source rather than binary form, which might be undesirable (for example, for intellectual property reasons);
- static error checking is in general precluded, and any errors are revealed only at instantiation time; moreover, error reports are typically verbose and unhelpful, because they relate to the consequences of a misuse rather than the misuse itself;
- there is a problem of 'code bloat', because different instantiations of the same templated code yield different units of binary code.

There is work being done to circumvent these problems by resorting to partial evaluation [39], but there is no immediate sign of a full resolution.

2.2 Container Types

A *container type* is a type of data structures whose purpose is to contain elements of another type, and to provide access to those elements. Examples include arrays, sequences, sets, associative mappings, and so on.

To a functional programmer, this looks like a *polymorphic datatype*; for example,

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

A data structure of type $\mathit{List}\ \alpha$ for some α will indeed contain elements of type α, and will (through pattern-matching, for example) provide access to them. Such polymorphic datatypes can be given a formal semantics via the categorical notion of a *functor* [10], an operation simultaneously on types (taking a type α to the type $\mathit{List}\ \alpha$) and functions (taking a function of type $\alpha \to \beta$ to the map function of type $\mathit{List}\ \alpha \to \mathit{List}\ \beta$).

However, that response is a little too simple. Certainly, some polymorphic datatypes and some functors correspond to container types, but not all do. For example, consider the polymorphic type

$$\textbf{data}\ \mathit{Transformer}\ \alpha = \mathit{Trans}\ (\alpha \to \alpha)$$

(The natural way to define this type in Haskell [34] is with a type synonym rather than a datatype declaration, but we've chosen the latter to make the point clearer.) There is no obvious sense in which a data structure of type $\mathit{Transformer}\ \alpha$ 'contains' elements of type α. Hoogendijk and de Moor [24] have shown that one wants to restrict attention to the functors with a *membership*

operation. Technically, in their relational setting, the membership of a functor F is the largest lax natural transformation from F to Id, the identity functor; informally, membership is a non-deterministic mapping selecting an arbitrary element from a container data structure. Some functors, such as *Transformer*, have no membership operation, and so do not correspond to container types according to this definition.

2.3 Iterators

The essence of the STL is the notion of an *iterator*, which is essentially an abstraction of a pointer. The elements of a container data structure are made accessible by providing iterators over them; the container typically provides operations *begin*() and *end*() to yield pointers to the first element and to 'one step beyond' the last element.

Basic iterators may be compared for equality, dereferenced and incremented. But there are many different varieties of iterator: *input iterators* may be dereferenced only as R-values (for reading), and *output iterators* only as L-values (for writing); *forward iterators* may be deferenced in both ways, and may also be copied (so that multiple elements of a data structure may be accessed at once); *bidirectional iterators* may also be decremented; and *random-access iterators* allow amortized constant-time access to arbitrary elements.

Despite the name, iterators in the STL do not express exactly the same idea as the ITERATOR design pattern, although they have the same intent of 'providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation' [17]. In fact, the proposed design in [17] is fairly close to an STL input iterator: an existing collection may be traversed from beginning to end, but the identities of the elements in the collection cannot be changed (although their state may be).

What all these varieties of iterator have in common, though, is that they point to individual elements of the data structure. This is inevitable given an imperative paradigm: as Austern [6] puts it, 'The moving finger writes, and having writ, moves on', and although under more refined iterator abstractions the moving finger may rewrite, and may move backwards as well as forwards, it is still a finger pointing at a single element of the data structure.

One functional analogue of iterators for traversing a data structure is the *map* operator that arises as the functorial action on element functions, acting on each element independently. More generally, one could point to *monadic maps* [15], which act on the elements one by one, using the monad to thread some 'state' through the computation.

However, lazy functional programmers are liberated by the availability of 'new kinds of glue' [26] for composing units of code, and have other options too. For example, they may use lists to achieve a similar separation of concerns: the interface between a collection data structure and its elements is via a list of these elements. The analogue to the distinction between input and output iterators (R-values and L-values) is the provision of one function to yield the *contents* of a

data structure as a list of elements, and another to *generate* a new data structure from a given list of elements.

This functional insight reveals a rather serious omission in the STL approach, namely that it only allows the programmer to manipulate a data structure in terms of its elements. This is a very small window through which to view the data structure itself. A map ignores the shape of a data structure, manipulating the elements but leaving the shape unchanged; iterator-style access also (deliberately) ignores the shape, flattening it to a list. Neither is adequate for capturing problems that exploit the shape of the data, such as pretty-printers, structure editors, transformation engines and so on. A more general framework is obtained by providing *folds* to consume data structures and *unfolds* to generate them [18] — indeed, the *contents* and *generate* functions mentioned above are instances of folds and unfolds respectively, and a *map* is both a fold and an unfold.

2.4 Concepts

We noted in the previous section that the essence of the STL is a hierarchy of varieties of iterator. In the STL, the members of this hierarchy are called *concepts*. Roughly speaking, a concept is a set of requirements on a type (in terms of the operations that are available, the laws they satisfy, and the asymptotic complexities in time and space); equivalently, a concept can be thought of as the set of all types satisfying those requirements.

Concepts are not part of C++; they are merely an artifact of the STL. An STL reference manual [6] can do no more than to describe a concept in prose. Consequently, it is a matter of informal argument rather than formal reasoning whether a given type is or is not a model of a particular concept. This is a problem for users of the STL, because it is easy to make mistakes by using an inappropriate type in a particular context: the compiler cannot in general check the validity of a particular use, and tracking down errors can be tricky. There have been some valiant attempts to address this problem by programming idioms [36, 31] or static analysis [21], but ultimately the language seems to be a part of the problem here rather than a part of the solution.

The solution seems obvious to the Haskell programmer: use type classes [29]. A type class captures a set of requirements on a type, or equivalently it describes the set of types that satisfy those requirements. (Type classes are more than just interfaces: they can provide default implementations of operations too, and type class inference amounts to automatic selection of an implementation.) Type classes are only an approximation to the notion of a concept in the STL sense, because they can capture only the signatures of operations and not their extensional (laws) or intensional (complexity) semantics. However, they are statically checkable within the language, which is at least a step forwards: C++ concepts cannot even capture signatures formally. The Haskell collection class library Edison [11, 33] uses type classes formally in the same way that STL uses concepts informally.

2.5 Algorithms and Function Objects

The bulk of the STL, and indeed its whole raison d'être, is the family of generic *algorithms* over container types made possible by the notion of an iterator. These algorithms are general-purpose operations such as searching, sorting, comparing, copying, permuting, and so on. Iterators decouple the algorithms from the container types on which they operate: the algorithm is described in terms of an abstract iterator interface, and is then applicable to any container type on which an appropriate iterator is available.

There is no new insight provided by the algorithms per se; they arise as a natural consequence of the abstractions provided (whether informally as concepts or formally as type classes) to access the elements of container types. In the STL, algorithms are represented as function templates, parametrized by models of the appropriate iterator concept. To a Haskell programmer, algorithms in this sense correspond to functions with types qualified by a type class.

The remaining principle on which the STL is built is that of a *function object* (sometimes called a 'functor', but in a different sense that the functors of category theory). Function objects are used to encapsulate function parameters to algorithms; typical uses are for parametrizing a search function by a predicate indicating what to search for, or a sorting procedure by an ordering.

Function objects also yield no new insight to the functional programmer. In the STL, a function object is represented as an object with a single method which performs the function. This is essentially an instance of the STRATEGY design pattern [17]. To a functional programmer, of course, function objects are unnecessary: functions are first-class citizens of the language, and a function can be passed as a parameter directly.

3 Datatype Genericity

We propose a new paradigm for generic programming, which we have called *datatype-generic programming* (DGP). The essence of DGP is the parametrization of values (for example, of functions) by a *datatype*. We use the term 'datatype' here in the sense discussed in Section 2.2: a container type, or more formally a functor with a membership operation. For example, '*List*' is a datatype, whereas '*int*' is merely a type.

(Since a datatype is one type parametrized by another — 'lists of αs, for some type α' — and a datatype-generic program is a program parametrized in turn by such a type-parametrized type, we toyed briefly with the idea of describing our proposal as for a *'type-parametrized–type'—parametrized theory of programming*, or TPTPTP for short. But we decided that was a bit of a mouthful.)

3.1 An Example of DGP

Consider for example the parametrically polymorphic programs *maplist*,

$$
\begin{aligned}
&\mathit{maplist} :: (\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta \\
&\mathit{maplist}\ f\ \mathit{Nil} \qquad\quad = \mathit{Nil} \\
&\mathit{maplist}\ f\ (\mathit{Cons}\ a\ x) = \mathit{Cons}\ (f\ a)\ (\mathit{maplist}\ f\ x)
\end{aligned}
$$

and (for the appropriate definition of the *Tree* datatype) *maptree*,

$$
\begin{aligned}
&\mathit{maptree} :: (\alpha \to \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \\
&\mathit{maptree}\ f\ (\mathit{Tip}\ a) \quad = \mathit{Tip}\ (f\ a) \\
&\mathit{maptree}\ f\ (\mathit{Bin}\ x\ y) = \mathit{Bin}\ (\mathit{maptree}\ f\ x)\ (\mathit{maptree}\ f\ y)
\end{aligned}
$$

Both of these programs are already quite generic, in the sense that a single piece of code captures many different specific instances. However, the two programs are themselves clearly related, and a DGP language would allow their common features to be captured in a single definition *map*:

$$
\begin{aligned}
&\mathit{map}\langle \mathit{Unit} \rangle\ () \qquad\quad = () \\
&\mathit{map}\langle \mathit{Const}\ a \rangle\ x \qquad = x \\
&\mathit{map}\langle + \rangle\ f\ g\ (\mathit{Inl}\ u) = \mathit{Inl}\ (f\ u) \\
&\mathit{map}\langle + \rangle\ f\ g\ (\mathit{Inr}\ v) = \mathit{Inr}\ (g\ v) \\
&\mathit{map}\langle \times \rangle\ f\ g\ (u, v) \quad = (f\ u, g\ v)
\end{aligned}
$$

This single definition is parametrized by a datatype; in this case it is defined by structural induction over a grammar of datatypes. The two parametrically polymorphic programs are of course instances of this one datatype-generic program: $\mathit{maplist} = \mathit{map}\langle \mathit{List} \rangle$ and $\mathit{maptree} = \mathit{map}\langle \mathit{Tree} \rangle$.

At first glance, this looks rather like a generic algorithm that could have come from the STL, and indeed in this case that is a valid analogy to make: *map*-like operations can be expressed in the STL. However, the crucial difference is that DGP allows a program to *exploit* the shape of the data on which it operates. For example, one could write datatype-generic functions to encode a data structure as a bit string and to decode the bit string to regenerate the data structure [27]: the *shape* of the data structure is related to the *value* of the bitstring. A more sophisticated example involves Huet's 'Zipper' [25] for efficiently but purely functionally representing a tree with a cursor position; different types of tree require different types of zipper, and it is possible [1, 23] to write datatype-generic operations on the zipper: here, the shape of one data structure determines the shape of an auxilliary data structure in a rather complicated fashion. Neither of these examples are possible with the STL.

3.2 Isn't This Just...?

As argued above, the parametrization of programs by datatypes is not the same as *generic programming* in the STL sense. The latter allows *abstraction from* the shape of data, but not *exploitation of* the shape of data. Indeed, this is why we chose a new term 'DGP' instead of simply using 'GP': we would prefer the latter term, but feel that it has already been appropriated for a more specific use than we would like. (For example, one often sees definitions such as 'Generic programming is a methodology for program design and implementation that separates

data structures and algorithms through the use of abstract requirement specifications' [37, p19]. We feel that such definitions reduce generic programming to good old-fashioned abstraction.)

DGP is not the same thing as *meta-programming* in general, and template meta-programming in particular. Meta-programming is a matter of writing programs that generate or otherwise manipulate other programs. For example, C++ template meta-programs yield ordinary C++ code when instantiated (at least notionally, although the code so generated is typically never seen); they are not ordinary C++ programs in their own right. A meta-program for a given programming language is typically not a program written in that language, but one written in a meta-language that generates the object program when instantiated or executed. In contrast, a datatype-generic program is a program in its own right, written in (perhaps an enrichment of) the language of the object program.

Neither is DGP the same thing as polymorphism, in any technical sense we know. It is clearly not the same thing as ordinary *parametric polymorphism*, which allows one to write a single program that can manipulate both lists of integers and lists of characters, but does not allow one to write a single program that manipulates both lists of integers and trees of integers. We also believe (but have yet to study this in depth) that DGP is not the same thing as *higher-order parametric polymorphism* either, because in general the programs are not parametric in the functor parameter: if they were, they might manipulate the shape of data but could not compute with it, as with the encoding and decoding example cited above.

Nor is it the same thing as *dependently typed programming* [5], which is a matter of parametrizing types by values rather than values by types. Dependent types are very general and powerful, because they allow the types of values in the program to depend on other values computed by that program; but by the same token they rule out the possibility of most static checking. (A class template parametrized by a value rather than a type bears some resemblance to type dependent on a value, but in C++ the actual template parameters must be statically determined for instantiation at compile time, whereas dependent type theory requires no such separation of stages.) It would be interesting to try to develop a calculus of dependently typed programming, but that is a different project altogether, and a much harder one too.

Finally, DGP is not simply Generic Haskell [12], although the datatype-generic program for *map* we showed above is essentially a Generic Haskell program. The Generic Haskell project is concentrating on the design and implementation of a language that supports DGP, but is not directly addressing the problem of developing a calculus of such programs. Our project has strong connections with the Generic Haskell project, and we are looking forward to making contributions to the design based on our theory-driven insights, as the language is making contributions to the theory by posing the question of how it may be used. However, Generic Haskell is just one possible implementation technique for DGP.

4 Patterns of Software

A *design pattern* 'systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems' [17]. The intention is to capture best practice and experience in software design in order to facilitate the education of novices in what constitutes good designs, and the communication between experts about those good designs. The software patterns movement is based on the work of Christopher Alexander, who for over thirty years has been leading a similar movement in architecture [3, 2].

It could be argued that many of the patterns in [17] are idioms for mimicking DGP in languages that do not properly support such a feature. Because of the lack of proper language support, a pattern can generally do no better than to motivate, describe and exemplify an idiom: it can refer indirectly to the idiom, but not present the idiom directly as a formal construction. For example, the ITERATOR pattern shows how an algorithm that traverses the elements of a collection type can be decoupled from the collection itself, and so can work with new and unforeseen collection types; but for each such collection type an appropriate new ITERATOR class must be written. (The programmer may be assisted by the library, as in Java [20], or the language, as in C♯ [14], but still has to write something for each new collection type.) A language that supported DGP would allow the expression of a single datatype-generic program directly applicable to an arbitrary collection type: perhaps a function to yield the elements as a lazy list, or a *map* operation to transform each element of a collection.

The situation is no better with the STL than with design patterns. We argued above that iterators in the STL sense are more general than the ITERATOR pattern. Nevertheless, C++ provides no support for defining the iterator concept, so it too can only be referred to indirectly; and again, for every new collection type an appropriate implementation of the concept must be provided.

As another example, the VISITOR pattern [17] allows one to decouple a multivariant datatype (such as abstract syntax trees for a programming language) from the specific traversals to be performed over that datatype (such as type checking, pretty printing, and so on), allowing new traversals to be added without modifying and recompiling each of the datatype variants. However, each new datatype entails a new class of VISITOR, implemented according to the pattern. A DGP language would allow one to write a single datatype-generic traversal operator (such as a *fold*) once and for all multivariant datatypes.

(Alexandrescu [4] does present a 'nearly generic' definition of the VISITOR pattern using clever template meta-programming, but it relies on C++ macros, and still requires the foresight in designing the class hierarchy to insert a call to this macro in every class in the hierarchy that might be visited.)

It is sometimes said that patterns cannot be automated; anything that can be captured completely formally is too restricted to be a proper pattern. Alexander describes a pattern as giving 'the core of the solution to [a] problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice' [3]; Gamma et al. state that 'design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and

reused as is' [17]. We are sympathetic to the desire to ensure that patternity does not become a synonym for 'a good idea', but do not feel that that means we should give up on attempts to formalize patterns.

Alexander, in his foreword to Gabriel's book [16], hopes that the software patterns movement will yield 'programs which make you gasp because of their beauty'. We think that's a goal worth aiming for, however optimistically. We have yet to see a meta-programming framework that supports beautiful programming (although we confess to being impressed by the intricate possibilities of template meta-programming demonstrated by [4]), but we have high hopes that datatype-generic programs could be breathtakingly beautiful.

5 Future Plans

The DGP project is due to start around September 2003; the work outlined in this paper constitutes about a third of the total. One of the initial aims of this strand will be an investigation into the relationships between generic programming (as exhibited in libraries like the STL), structural and behavioural design patterns (as described by [17]), and the mathematics of program construction (epitomized by Hoogendijk and de Moor's categorical characterization of datatypes [24]).

In the short term, we intend to use the insights gained from this investigation to prototype a datatype-generic collection library in Generic Haskell [12] (perhaps as a refinement of Okasaki's Edison library [33]). This will allow us to replace type-unsafe meta-programming with type-safe and statically checkable datatype-generic programming. Ultimately, however, we hope to be able to apply these insights to programming in more traditional object-oriented languages, perhaps by compilation from a dedicated DGP language.

But the real purpose of the project will be to generalize theories of program calculation such as Bird and de Moor's relational 'algebra of programming' [10], to make it more applicable to deriving the kinds of programs that users of the STL write. This will link with Backhouse's strand of the DGP project, which is looking at more theoretical aspects of datatype genericity: higher-order naturality properties, logical relations, and so on. We intend to build on this work to develop a calculus for generic programming.

More tangentially, we have been intrigued by similarities between some of the more esoteric techniques for template meta-programming [13, 4] and some surprising possibilities for computing with type classes in Haskell [32, 30, 9]. It isn't clear yet whether those similarities are a coincidence or evidence of some deeper correspondence; in the light of our arguments in this paper that type classes are the Haskell analogue of STL concepts, we suspect there may be some deep connection here.

6 Acknowledgements

The help of the following people and organizations is gratefully acknowledged:

– Roland Backhouse, Graham Hutton, Ralf Hinze and Johan Jeuring, for their contributions to the DGP grant proposal;
– Richard Bird, for inspiring and encouraging this line of enquiry;
– Tim Sheard, for his elegant definition of generic programming;
– EPSRC grant GR/S27078/01, for financial support.

References

[1] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In Martin Hofmann, editor, *LNCS 2701: Typed Lambda Calculi and Applications*, pages 16–30. Springer-Verlag, 2003.
[2] Christopher Alexander. *The Nature of Order*. Oxford University Press, To appear in 2003.
[3] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.
[4] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.
[5] Lennart Augustsson. Cayenne: A language with dependent types. *SIGPLAN Notices*, 34(1):239–250, 1999.
[6] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.
[7] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.
[8] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Bernhard Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.
[9] Roland Backhouse and Jeremy Gibbons. Programming with type classes. Presentation at WG2.1#55, Bolivia, January 2001.
[10] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
[11] Andrew Bromage. Haskell Foundation Library. www.sourceforge.net/projects/hfl/, 2002.
[12] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Universiteit Utrecht, 2001.
[13] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
[14] Peter Drayton, Ben Albahari, and Ted Neward. *C♯ in a Nutshell*. O'Reilly, 2002.
[15] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Dept INF, Univ Twente, June 1994.
[16] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.
[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[18] Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Palgrave, 2003.
[19] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming*. Kluwer Academic Publishers, 2003.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
[21] Douglas Gregor and Sybille Schupp. Making the usage of STL safe. In Gibbons and Jeuring [19].
[22] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002. Earlier version appears in LNCS 1837: Mathematics of Program Construction, 2000.
[23] Ralf Hinze and Johan Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, 2001.
[24] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.
[25] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.
[26] John Hughes. Why functional programming matters. *Computer Journal*, 1989.
[27] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–72, 2002.
[28] Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors. *Generic Programming*. Springer-Verlag, 2000.
[29] Mark P. Jones. *Qualified Types: Theory and Practice*. DPhil thesis, University of Oxford, 1992.
[30] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, 2002.
[31] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.
[32] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. In *Symposium on Principles of Programming Languages*, pages 233–244, 2002.
[33] Chris Okasaki. An overview of Edison. Haskell Workshop, 2000.
[34] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
[35] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, 2002.
[36] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.
[37] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.
[38] Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.
[39] Todd Veldhuizen. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming*, October 2000.

Unifying Tables, Objects and Documents

Erik Meijer and Wolfram Schulte

Microsoft Corporation

Abstract. This paper proposes a number of type-system and language extensions to natively support relational and hierarchical data within a statically typed object-oriented setting. In our approach SQL tables and XML documents become first class citizens that benefit from the full range of features available in a modern programming language like $C^\sharp$ or Java. This allows objects, tables and documents to be constructed, loaded, passed, transformed, updated, and queried in a unified and type-safe manner.

1 Introduction

The most important current open problem in programming language research is to increase programmers productivity, that is to make it easier and faster to write correct programs [38]. The integration of data access in mainstream programming languages is of particular importance — millions of programmers struggle with this every day. Data sources and sinks are typically XML documents and SQL tables, but they don't merge nicely into a statically typed object-oriented setting in which most production software is written.

This paper addresses how to integrate tables and documents into modern object-oriented languages by providing a novel type-system and corresponding language extensions.

1.1 The Need for a Unification

Distributed web-based applications are predominantly structured using a three-tier model that most commonly consists of a *middle tier* containing the business logic that extracts relational data from a *data services tier* and munches it into hierarchical data that is displayed in the *user interface tier*. The middle tier is often programmed in an object-oriented language such as Java or $C^\sharp$.

As a consequence, middle tier programs have to deal with relational data (SQL tables), object graphs, and hierarchical data (HTML, XML). Unfortunately these three different worlds are not very well integrated. As the following ADO.Net based example shows, access to a database in this style involves sending a string representation of a SQL query over an explicit connection via a stateful API and then iterating over a weakly typed representation of the result set:

```
SqlConnection Conn = new SqlConnection(...);
SqlCommand Cmd = new SqlCommand("SELECT Name,HP FROM Pokedex",Conn);
Conn.Open();
SqlDataReader Rdr = Cmd.ExecuteReader();
```

Creating HTML or XML documents is then done by emitting document fragments in string form, without separating the model and presentation:

```
while (Rdr.Read()) {
 Response.Write("<tr><td>");
 Response.Write(Rdr.GetInt32(0));
 Response.Write("</td><td>");
 Response.Write(Rdr.GetString(1));
 Response.Write("</td></tr>");
}
```

Communication between the different tiers using untyped strings is obviously very brittle with lots of opportunities for errors and zero probability for static checking. The cynical thing is that due to the poor integration, performance suffers badly as well.

The next code fragment rewrites the same functionality using a hypothetical language that unifies objects, tables and documents.

```
tr* pokemon =
  select <tr>
          <td>{Name}</td><td>{HP}</td>
         </tr>
  from Pokedex;

Table t =
 <table>
   <tr><th>Name</th><th>HP</th></tr>
   {pokemon}
 </table>;

Response.Write(t);
```

In this case, strongly typed XML values are first-class citizens (i.e. the XML literal `<table>...</table>` has type static `Table`) and SQL-style `select` queries are build-in. There is ample opportunity for static checking, and because the SQL and XML type-systems are integrated into the language, the compiler can do a better job in generating efficient code.

1.2 Growing a Language

It is easy to criticize the current lack of integration between tables, objects and documents, but it is much harder to come up with a design that gracefully unifies

these separate worlds. No main-stream programming language has yet emerged that realizes this vision [7].

Often language integration only deals with SQL *or* with XML, but usually not with both [12, 26, 15, 19, 2, 11, 29]. Alternatively they start from a completely new language such as XQuery, or XDuce or CDuce [6, 24, 44, 10]. Approaches based on language binding using some kind of pre-compiler such as XSD.exe, Castor, or JAXB [31, 1] do not achieve a real semantic integration. The impedance mismatch between the different type-systems then leads to strange anomalies or unnatural mappings. Another popular route to integrate XML and SQL is by means of domain specific embedded languages [25] using functional language such as Scheme or Haskell [35, 36, 33, 34, 30, 27, 20, 42, 46, 12] as the host. In our experience however, the embedded DSL approach does not scale very well, and it is particularly difficult to encode the domain specific type-systems [40] and syntax into the host language.

In his invited talk at OOPSLA98 [22], Guy Steele remarked that

> ... from now on, a main goal in designing a language should be to plan for growth. The language should start small, and the language must grow as the set of users grows.

This paper shows how to grow a modern object-oriented language (we take $C^\sharp$ as the host language, but the same approach will work with Java, Visual Basic, C++, etc.) to encompass the worlds of tables and documents by adding new types and expressions. In the remainder of this paper we will discuss:

Streams (Section 2) Streams are homogenous sequences of values of variable length. A database table consists of zero or more tuples; in the document world nodes can have zero or more sub-documents of the same kind, and in the object world we often work with (lazy) streams of values.

Tuples (Section 3) Tuples are heterogeneous sequences of values of fixed length. As we have just noticed, a database table is a stream of tuples; in the document world the `sequence` construct is used to model groups of sub-documents that must be present in a particular order, and finally several proposals have been made to extend Java and other object-oriented languages with tuples [43, 28].

Unions (Section 4) Unions represent a choice between values of different type. They play a very important role in semi-structured documents [8] and many schemas use the `choice` construct to model alternatives. Union types also occur naturally in the result-types of queries.

Content Classes (Section 5) Content classes are ordinary classes whose members can be anonymous (unnamed). We use content classes to model top-level elements and complex types in document schemas.

Queries (Section 6) Finally we will extend our repertoire of accessors of our new types to match the expressive power of XPath and SQL queries. These accessors include implicit (homomorphic extension) and explicit (apply-to-all) mapping over streams, filtering, transitive member access, and relational select and join.

The growth of our experimental language is controlled by applying the following design principles:

Denotables values should be (easily) expressible If programmers can declare a variable of a certain type, it must be possible to write an expression of that type in a convenient way.

Expressible values should be denotable If programmmers can write an expression of a certain type, it must be possible to declare a variable whose static type precisely matches that of the expression.

No forced identity Programmers should never be forced to introduce either nominal identity of types, or object identity of values (aliasing).

Orthogonality There should be no special cases that discriminate between tables, documents and objects. Operations should work uniformly across the three worlds.

Flexibility The new types should have rich subtyping relationships that ease in writing type correct and evolvable software [9].

2 Streams

Streams are *generically typed refinements of iterators*, the pair of twin interfaces `IEnumerable` and `IEnumerator` in $C^\sharp$, or the corresponding `Iterator` interface in Java. Iterators encapsulate the logic for enumerating elements of collections.

The `foreach` loop of $C^\sharp$ makes it very convenient to *consume* values of type `IEnumerable` (future versions of Java will have a similar construct). For instance, since type `string` implements the `IEnumerable` interface, we can iterate over all the characters in a string using a simple `foreach` loop:

```
foreach(char c in s) Console.WriteLine(c);
```

The `foreach` loop in $C^\sharp$ is syntactic sugar for the following (simplified) `while` loop that calls into the `IEnumerable` and `IEnumerator` interfaces:

```
IEnumerator e = ((IEnumerable)s).GetEnumerator();
while (e.MoveNext()) { char c = (char)e.Current;
  Console.WriteLine(c);
}
```

While consuming an iterator is easy, it is much more difficult to write a generator that *implements* the `IEnumerable` (or the underlying `IEnumerator`) interface. In order to implement the `IEnumerable` interface on type `string` for instance, we have to manually create a state-machine that iterates over the individual characters in the string via `MoveNext` and exposes the current character via the `Current` property:

```
class string: IEnumerable {

  IEnumerator GetEnumerator() { return new Chars(this); }
```

```
    private class Chars : IEnumerator {
      private string s; private int i = 0; private char c;

      Chars(string s) { this.s = s; }

      public bool MoveNext() {
        if (i < s.Length) {
          c = s[i++]; return true;
        } else {
          return false;
        }
      }

      public char Current { get { return c; } }
    }
  }
```

Note that this implementation does not correctly handle the extreme cases of calling `Current` before the first call to `GetNext` and calling it after `GetNext` has returned `false`.

In $C^\sharp$ and Java iterators are denotable, but not easily expressible. Moreover, the type `IEnumerable` is not very accurate since it does not convey the element type of the iterator. In other words, iterators of a particular type are expressible, but not precisely denotable.

We remedy both problems by introducing a new type of streams and a new statement to generate streams:

- The type T`*` denotes homogenous streams of arbitrary length with elements of type T. Type T`*` is a subtype of both `IEnumerable` and `IEnumerator`.
- Stream generators are like ordinary methods except that they may yield multiple values instead of returning a single time. The `yield` e statement returns the value of expression e into the `Current` property of its corresponding stream and suspends execution until `MoveNext` is called at which time execution resumes. Upon termination of the iterator `MoveNext` returns `false`.

Using streams and generators it becomes much simpler to enumerate all the characters in a string. The helper method `char* explode(string s)` generates the stream of the individual characters of string `s`. The `GetEnumerator` method of class `string` then simply explodes itself:

```
  class string: IEnumerable {

    public IEnumerator GetEnumerator() { return this.explode(); }

    private char* explode() {
```

```
    int e = this.Length; for(int i = 0; i < e; i++) yield s[i];
  };
}
```

In this case maintaining the state is implicit in the control-flow of the `explode` function and in particular the borderline cases are handled correctly by definition.

Streams and generators are not new concepts. They are supported by a wide range of languages in various forms [21, 5, 39, 28, 32, 37], and in particular future versions of $C^\sharp$ will also support iterators. Our approach is a little different in that:

- We classify streams into a hierarchy of streams of different length (`!`, `?`, `+`, `*`, see below).
- We automatically flatten streams of streams (see Section 2.2).
- Our streams are covariant (see below).
- We identify the value `null` with the empty stream (see Section 2.1).

To keep type-checking tractable, we restrict ourselves to the following four stream types: T`*` denotes possibly empty and unbounded streams with elements of type T, T`+` denotes non-empty possibly unbounded streams with elements of type T, T`?` denotes streams of at most one element of type T, and T`!` denotes streams with exactly one element of type T. We will use T`?` to represent optional values, where the nonexistence is represented by the value `null` and analogously we use T`!` to represent non-null values.

The different stream types form a natural subtype hierarchy, where subtyping corresponds to stream inclusion:

$$\begin{aligned} T\texttt{!} &<: T\texttt{+} \\ T\texttt{+} &<: T\texttt{*} \\ T\texttt{?} &<: T\texttt{*} \end{aligned}$$

For instance the subtype relation T`!` $<: T$`+` reflects the fact that a stream of exactly one element is also a stream of at least one element.

We embed non-stream types T into the hierarchy by placing them between non-null values T`!` and possibly null values T`?`:

$$\begin{aligned} T\texttt{!} &<: T \\ T &<: T\texttt{?} \end{aligned}$$

This inclusion allows programmers to precisely state their intentions with respect to `null` values: T`!` means `null` is not allowed, T`?` means `null` is expected, and T means `null` is exceptional.

The next two rules reflect the facts that `null` (we use Ø`?` for the null-type) is a possible value of any reference type, but that value types are never `null`:

$$\begin{aligned} \text{Ø}\texttt{?} &<: T, \quad T \text{ is a reference type} \\ T &<: T\texttt{!}, \quad T \text{ is a value type} \end{aligned}$$

Like arrays, streams are covariant. This means that subtyping on the element types is lifted to subtyping on streams. The special case for the `null` type says that possibly-null values can be `null`:

$$\frac{S <: T}{S* <: T*}$$

$$\emptyset? <: T?$$

Let `Button` be a subtype of `Control`, then the first rule says that `Button*` is a subtype of a stream of controls `Control*`. The second rule says for instance that `null` can be assigned to a variable of type `int?`.

2.1 Nullness

The type T`!` denotes streams with exactly one element, and since we identify `null` with the empty stream, this implies that values of type T`!` can never be `null`. Dually, the type T`?` denotes streams with either zero (that is `null`) or exactly one element.

Values of type T`?` model the explicit notion of nullability as found in SQL by providing a standard implementation of the `null` design pattern [23]; when a receiver of type T`?` is `null`, accessing any of its members returns `null` instead of throwing an exception as in $C^\sharp$ or Java:

```
string? t = null;
int? n = t.Length; // n = null
```

In Objective-C [3] this is the standard behavior for any receiver object that can be `null`. In section 6.1 we show how member-access is lifted over streams in general.

Being able to express that a value *cannot* be `null` via the type system allows static checking for `null` pointers (see [16, 18] for more examples). This turns many (potentially unhandled) dynamic errors into compile-time errors.

One of the several methods in the .NET base class library that throws an `ArgumentNullException` when its argument is `null` is the `IPAddress.Parse` function. Consequently, the implementation of `IPAddress.Parse` needs an explicit `null` check:

```
public static IPAddress Parse(string ipString) {
  if (ipString == null)
    throw new ArgumentNullException("ipString");
  ...
}
```

Dually, clients of `IPAddress.Parse` must be prepared to catch and deal with a possible `ArgumentNullException`. Nothing of this is apparent in the type of the

`Parse` method in $C^\sharp$. In Java at least the signature of `Parse` would show that it possibly throws an exception.

It would be much cleaner if the type of `IPAddress.Parse` indicated that it expects its `string` argument to be non-`null`:

```
public static IPAddress Parse(string! a);
```

Now, the type-checker statically rejects any attempt to pass a string that might be `null` to `IPAddress.Parse`.

The proof obligation for returning a non-null stream T`!` or T`+` is similar to proving the definite assignment rule in $C^\sharp$ or Java. For statement blocks that return or yield non-empty streams, each non-exceptional execution path should return or yield at least one non-null value. The type-checker will therefore accept the first definition of `FromTo` but will reject the second:

```
int+ FromTo(int s, int d, int e) {
  yield s; while(s <= e) yield s += d;
}

// Type error
int+ FromTo(int s, int d, int e) {
  while(s <= e){ yield s; s += d; }
}
```

Non-empty streams T`+` are implicitly convertible to possibly empty streams T`*`; we can forget the fact that a stream has at least one element. It is in general *not* safe to downcast from a possibly empty stream T`*` to a non-empty stream T`+`. At first sight we might think that testing if the stream contains at least one non-null value would suffice. Alas this is not true. By cunningly using side-effects, the generator function `OnlyOnce()` only yields `4711` the first time it is evaluated and every subsequent evaluation produces an empty stream:

```
bool Done = false;
int* OnlyOnce() {
  if(!Done){ Done = true; yield 4711; }
};
int+ xs = (int+)OnlyOnce();   // 1. cast succeeds
int+ xs = (int+)OnlyOnce();   // 2. cast fails
```

To prevent such loopholes, down casting from T`*` to T`+` will only succeed if the dynamic type of the underlying stream is T`+`.

2.2 Flattening

We have to be very careful to ensure that every value in a (nested) stream is yielded at most once, otherwise we might end up with a quadratic instead of a linear number of yields when generating certain (recursive) streams [45]. For instance this happens if a nested stream like `[[...[[[],0],1],...],n-1]` gets

recursively flattened into the non-nested stream `[0,1,...,n-1]` as in the next example:

```
// Iota(n) generates the stream [0,1,..,n-1]
int* Iota(int n){
  if(n>0){
    foreach(int i in Iota(--n)) yield i;
    yield n;
  }
}
```

Note that we are forced to flatten the stream produced by the recursive invocation of `Iota(n)` to generate a stream of the required type `int*`. Apart from these typing issues, there is absolutely no reason that the actual instance of a nested stream should be flattened since we can easily iterate over the leaf elements (the *yield*) of a nested stream.

So all that is required to type-check generators of nested streams is to flatten the *type* of a stream, which again does not imply that the underlying implementation of streams gets flattened as well. Table 2.2 gives the general flattening rules for nested streams T^{ij} of all possible combinations of stream constructors: The

T^{ij}	j = !	?	+	*
i = !	!	?	+	*
?	!	?	+	*
+	+	*	+	*
*	+	*	+	*

Fig. 1. Flattening rules for streams

congruence T`*+` $= T$`+`, for instance, reflects the fact that a non-empty stream of possibly empty stream flattens into a non-empty stream, while T`+*` $= T$`*` reflects that a possibly empty stream of non-empty streams flattens to a possibly empty stream.

Using the flattening rules, we can now write a linear time version of the `Iota` function that returns a nested stream of streams of type `int*`:

```
// Iota(n) generates the stream [[...[[[],0],1],...],n-1]
int* Iota(int n){
  if(n>0){
    yield Iota(--n); yield n;
  }
}
```

3 Tuples

Tuples are *heterogeneous* sequences of *optionally labelled* values of *fixed* length. Another way of viewing tuples is as anonymous structs whose members are ordered, in particular tuples have no object identity.

The function `DivMod` returns the quotient and remainder of its arguments as a tuple that contains two named integer fields `sequence{int Div, Mod;}`:

```
sequence{int Div, Mod;} DivMod(int x, int y) {
  return new(Div = x/y, Mod = x%y);
}
```

The members of a tuple do not need to be labelled, for example, we can create a tuple consisting of a labelled `Button` and an unlabelled `string` as follows:

```
sequence{Button b; string;} x = new(b=new Button(), "OK");
```

An unlabelled member of a *nominal* type is a shorthand for the same member implicitly labelled with its type.

Tuples can be picked apart constant indexers, `DivMod(47,11)[0]` for instance selects `47`, or by named member access, provided of course that tuple has a member m, for instance `x.b`.

3.1 Subtyping

Like streams, tuples are subject to a rich subtype hierarchy. The first subtype relation for tuples formalizes the fact that labels are optional and that we can forget them by upcasting:

$$\texttt{sequence}\{\ldots;\ T\ m\,;\ldots\} <: \texttt{sequence}\{\ldots;\ T\ ;\ \ldots\}$$

Using this rule we see that we can assign `DivMod(47,11)` to a an unlabelled pair of integers of type `sequence{int; int;}`.

We can forget the ordering, nesting, and labels of a tuple by upcasting a tuple to a stream. The special cases give tighter types for the empty tuple (which gets converted to the empty stream `null`) and singleton tuple (which gets converted to its underlying value):

$$\begin{aligned}
&\texttt{sequence}\{\} <: \varnothing\texttt{?}\\
&\texttt{sequence}\{T\} <: T\\
&\texttt{sequence}\{\ldots;\ T\ ;\ \ldots\} <: \texttt{choice}\{\ldots;T\ ;\ldots\}\texttt{*}
\end{aligned}$$

Using the last conversion, we can enumerate the values of any tuple as a stream, i.e. the tuple `new(4711, true, 'z', 3.14)` can be converted into the stream `[4711, true, 'z', 3.14]` of type `choice{int; bool; char; float}*`.

3.2 Non-Nullness for tuples

Even though tuples have no object identity, the fact that they are convertible to streams makes them subtly different from nominal value types.

Suppose that we would add the rule that tuples are not `null`, i.e., `sequence{...}` <: `sequence{...}!`. Then by applying this rule in combination with the singleton rule `sequence{`T`}` <: T we could assign the value `null` to a variable of non-null type `Button!`:

```
// Type error
sequence{Button;}  a = new(null);
sequence{Button;}! b = a;
Button!            c = b; // c = null
```

To maintain type-soundness soundness we have a weaker rule that states that a tuple is non-null if it has at least one member that is non-null. This guarantees that when the tuple is converted to a stream the resulting stream has the right cardinality. For singleton sequences the conversion also holds in the reverse direction:

`sequence{...;` T`!` m`; ...}` <: `sequence{...;` T m`; ...}!`
`sequence{`T m`;}!` <: `sequence{`T`!` m`;}`

By applying this rule in combination with the fact that `int <: int!`, we can show that the sequence of integers `new(1)` is convertible into a non-empty stream of type `sequence{int ;}` <: `sequence{int! ;}` <: `sequence{int ;}!` <: `int*!` <: `int+`.

3.3 Streams+Tuples = Tables

Relational data is stored in tables, which are sets of tuples. Sets can be represented by streams, thus streams and tuples together can be used to model relational data.

The table below contains some basic facts about Pokemon characters such as their name, their strength, their kind, and the Pokemon from which they evolved (see `http://www.pokemon.com/pokedex/` for more details about these interesting creatures).

Name	HP	Kind	Evolved
Meowth	50	Normal	
Rapidash	70	Fire	Ponyta
Charmelon	80	Fire	Charmander
Zubat	40	Plant	
Poliwag	40	Water	
Weepinbell	70	Plant	Bellsprout
Ponyta	40	Fire	

Each row in this table is a value of type `Pokemon` and the table itself is modelled as a variable `Pokedex` of type `Pokemon*`. The keyword `type` identifies the name on the left with the type expression on the right. It is just an abbreviation mechanism.

```
enum Kind {Water, Fire, Plant, Normal, Rock}

type Pokemon = sequence{
  string Name; int HP; Kind Kind; string? Evolved;
}

Pokemon* Pokedex;
```

The fact that basic Pokemon are not evolutions of other Pokemon shows up in that the `Evolved` column has type `string?`.

Representing tables is necessary for the integration of relational data, but it is not sufficient: we also have to provide operations that work on tables. We will introduce such query expressions in Section 6.3.

4 Unions

Union types often appear in content classes (see section 5 below). The type `Address` uses a union type `choice{ string Street; int POBox; }` to allow either a member `Street` of type `string` or a member `POBox` of type `int` as part of an `Address`:

```
class Address {
  sequence{
    choice{ string Street; int POBox; };
    string City; string? State; int Zip;
    string Country;
  };
}
```

The second situation in which union types are used is in the result types of generalized member access (see Section 6). For example, when variable `p` has type `Pokemon`, the expression `p.*` returns a stream containing all the members of a `Pokemon` instance which has type `choice{string; int; Kind; string?}*`. Using the subtype rules for `choice` and streams given below, we can show that this is isomorphic to `choice{string; int; Kind;}*`.

We can inject any type T into a union containing that type; singleton labelled tuples are injected into labelled unions:

$$T \mathrel{<:} \texttt{choice}\{T\,;\,\ldots\}$$
$$\texttt{sequence}\{T\ m\} \mathrel{<:} \texttt{choice}\{T\ m\,;\,\ldots\}$$

Except for boxing, `choice{...}` <: `object`, there is no implicit elimination rule for union types. In other words, `choice{`T ; S`}` is an upperbound for S and T, but not a least upperbound. The reason is that we do not consider `Control` and `choice{Button; Control;}` to be isomorphic, which would be the case with a least upperbound interpretation.

Choice types are idempotent (duplicates are removed), and associative and commutative (nesting and order of members are ignored):

$$\texttt{choice}\{\ldots; F; F; \ldots\} = \texttt{choice}\{\ldots; F; \ldots\}$$
$$\texttt{choice}\{\ldots; \texttt{choice}\{\ldots\}; \ldots\} = \texttt{choice}\{\ldots; \ldots; \ldots\}$$
$$\texttt{choice}\{\ldots; F; G; \ldots\} = \texttt{choice}\{\ldots; G; F; \ldots\}$$

Streams distribute over unions. Non-nullness and possibly nullness distribute in both ways, and any inner streams gets absorbed by an outer `+` or `*`:

$$\texttt{choice}\{\ldots; T; \ldots\}\texttt{!} = \texttt{choice}\{\ldots; T\texttt{!}; \ldots\}\texttt{!}$$
$$\texttt{choice}\{\ldots; T; \ldots\}\texttt{?} = \texttt{choice}\{\ldots; T\texttt{?}; \ldots\}\texttt{?}$$
$$\texttt{choice}\{\ldots; T^i; \ldots\}\texttt{+} = \texttt{choice}\{\ldots; T; \ldots\}\texttt{+}$$
$$\texttt{choice}\{\ldots; T^i; \ldots\}\texttt{*} = \texttt{choice}\{\ldots; T; \ldots\}\texttt{*}$$

where i is any stream functor.

The flattening and distribution rules allow us to normalize streams of choices: inner stream functors can either be eliminated completely or can be moved out of the choice.

5 Content Classes, XSDs and XML

Now that we have introduced streams, tuples, and unions, our type system is rich enough to model a large part of the XSD schema language [17]; our aim is to cover as much of the essence of XSD [41] as possible while avoiding most of its complexity.

The correspondence between XSD particles such as `<sequence>` and `<choice>` with local element declarations and the type constructors `sequence` and `choice` with (labelled) fields should be intuitively clear. Likewise, the relationship of XSD particles with occurrence constraints to streams is unmistakable. For T`*` the attribute pair `(minOccurs, maxOccurs)` is `(0, unbounded)`, for T`+` it is `(1, unbounded)`, for T`?` it is `(0, 1)`, and for T`!` it is `(1,1)`.

The content class `Address` that we defined in Section 4 corresponds to the following XSD schema `Address`:

```
<element name="Address">
  <complexType>
    <sequence>
      <choice>
        <element name="Street" type="string">
```

```
        <element name="POBox" type="integer">
      </choice>
      <element name="City" type="string">
      <element name="State" type="string" minOccurs="0"/>
      <element name="Zip" type="integer"/>
      <element name="Country" type="string"/>
    </sequence>
  </complexType>
</element>
```

The only difference between a content class and a normal $C^\sharp$ class is the fact that the members of content class can be unlabelled (just like the members of tuples and unions). As a consequence, unlabelled content can only ever be accessed via its individually named children, which allows the compiler to choose the most efficient data layout.

The next example schema defines two top level elements `Author` and `Book` where `Book` elements can have zero or more `Author` members:

```
<element name="Author">
  <complexType>
    <sequence>
      <element name="Name" type="string"/>
    </sequence>
  </complexType>
</element>

<element name="Book">
  <complexType>
    <sequence>
      <element name="Title" type="string"/>
      <element ref="Author" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>
```

In this case, the local element reference is modelled by an unlabelled field and the schema is mapped onto the following two content type declarations:

```
class Author { string Name; }
class Book { sequence{ string Title; Author*; } }
```

All groups such as the one used in the following schema for the complex type `Name`

```
<complexType name="Name">
  <all>
    <element name="First" type="string"/>
    <element name="Last" type="string"/>
```

```
    </all>
  </complexType>
```

are mapped to ordinary fields of the containing type, i.e. without a `sequence`:

```
class Name { string First; string Last; }
```

As these examples show, both top-level element declarations and named complex type declarations are mapped to top-level types. This allows us to unify derivation of complex types and substitution groups of elements using standard inheritance.

5.1 XML Literals

XML literals are an intuitive way to construct instances of content classes by making XML serialization into a first class language construct. For example, we can define an `Address` instance by directly assigning an XML document that confirms to the schema for `Address` as follows:

```
Address Microsoft =
  <Address>
    <Street>One Microsoft Way</Street>
    <City>Redmond</City><Zip>98052</Zip>
    <Country>USA</Country>
  </Address>;
```

XML literals can also have placeholders to describe *dynamic* content (similar to anti-quoting as found in Lisp and other languages). We use the XQuery [6] convention whereby an arbitrary expression or statement block can be embedded inside an element by escaping it with curly braces:

```
Author NewAuthor(string name) {
  return <Author>{name.ToUpper()}</Author>;
}
```

Embedded expressions must return or yield values of the required type (in this case `string`). Validation of XML literals with placeholders is non-trivial and is the subject of a forthcoming paper.

XML literals are just object constructors, there is nothing special about content classes. Hence we can write XML literals to construct values of *any* type, for example, the next assignment creates an instance of the standard `Button` class and sets its `Text` field to the string `"Click Me"`:

```
Button b = <Button>
             <Text>Click Me</Text>
           </Button>;
```

6 Generalized Member Access

In the previous sections we have concentrated on the type-system extensions to our hypothetical programming language. This section extends our repertoire of *expressions* to transform and query values of these new types.

6.1 Map, Filter, Fold

To make the creation of streams as concise as possible, we allow statement blocks (anonymous method bodies) as expressions. In the example below we assign the (lazy) infinite stream of positive integers to the variable `nats` by using an anonymous method body as an expression:

```
// block expression that yields the stream [0,1,2,...]
int* nats = { int i=0; while(true) yield i++; };
```

Our stream constructors (`*`, `+`, `?`, `!`) are functors, and hence we *implicitely lift member access on the element type of a stream over the stream itself*. For instance, to convert each individual string in a stream `Ss` of strings to uppercase, we can simple write `ss.ToUpper()`:

```
string* Ss = { yield "Hello"; yield "World!"; };
string* SS = Ss.ToUpper();
```

If both the stream and its elements have the same member no lifting takes place, and member access on the whole stream is the best match. For example, since `GetType()` is defined for both `string` and `string*`, the expression `Ss.GetType()` will return the dynamic type of the stream `Ss`.

If we nevertheless want to lift member access over a stream, we can use an *apply-to-all* block. For example, to get all the dynamic types of the elements of a stream we write `Ss.{ return it.GetType(); }`. The implicit argument `it` inside the apply-to-all block plays a similar role as the implicit argument `this` for methods and refers successively to each element of the stream `nats`.

As the next example shows, the apply-to-all block itself can yield a stream, in which case the resulting nested stream is flattened according to the rules of table 2.2:

```
// self-counting numbers: 1, 2,2, 3,3,3, 4,4,4,4, ...
int* rs = nats.{ for(i=1; i<it; i++) yield it; };
```

If an *apply-to-all* block returns `void`, no new stream is constructed and the block is eagerly applied to all elements of the stream. For example to print all the elements of a stream we can just map `Console.WriteLine` over each element:

```
nats.{ Console.WriteLine(it); };
```

Apply-to-all blocks can be stateful, so we can use them to do reductions or folds. For example, we can sum all integers in an integer stream `xs` by adding each element of the stream to a local variable `s`:

```
int sum(int* xs){
  int s = 0;
  xs.{ s += it; return; };
  return s;
}
```

Note that we need the `return` statement inside the block to ensure that the return type of the block is `void` such that the iteration is performed eagerly.

Often we want to filter a stream according to some predicate on the elements of the stream. For example, to construct a stream with only odd numbers, we filter out all even numbers from the stream `nats` of natural numbers using the filter expression

```
int* odds1 = nats[it%2 == 1];
```

For each element in the stream to be filtered, the predicate is evaluated with that element as `it`. Only if the predicate is true the element becomes part of the new stream.

On closer inspection, we realize that filters are just abbreviations of an apply-to-all-block:

```
int* odds2 = nats.{if (it%2 == 1) return it;};
```

Hence `odds1` and `odds2` denote streams that both have the same elements in the same order.

Lifting over non-null types is different from lifting over the other stream types, since the fact that the receiver object is not `null` does not imply that its members are not `null` either. For example when we create a new non-null `Button` instance using the default constructor, it's `Parent` field will definitively be null:

```
Button! b = <Button/>;
Control p = b.Parent; // Parent is null
```

Hence the return type of lifting over a non-null type is not guaranteed to return a non-null type.

The table 6.1 show how lifting of member-access interacts with streams types. Let T^j be a stream type, and m of type S^i be a member of the element type T that we want to lift over the stream . The result type of lifting m is then given by $s^{i \oplus j}$ (here _ denotes a non stream type):

Member access is not only lifted over streams, but over all structural types. For example the expression `xs.x` will return the stream `true, 1, 2` of union type `choice{bool; int;}+` when `xs` is defined as:

```
sequence{ bool x; sequence{ int x; }*; } xs =
  new( x=true, { yield new(x=1); yield new(x=2); } );
```

Lifting over union types introduces a possibility of nullness for members that are not in all of the alternatives.

$\oplus$	j=_	!	?	+	*
i=_		?	?	*	*
!	!	!	?	*	*
?	?	?	?	*	*
+	+	+	*	+	*
*	*	*	*	*	*

Fig. 2. Lifting over streams

Suppose `x` has type `choice{ int; string; }`. Since only `string` has a `Length` member, the type of `x.Length` is `int?` which reflects the fact that in case the dynamic type of `x` is `int`, the result of `x.Length` will be `null`. Since `int` and `string` both have a member `GetType()`, the return type of `x.GetType()` is `Type`:

```
choice{ int; string; } x = 4711;
int? n = x.Length;    // null
Type t = x.GetType(); // System.Int32
```

In case the alternatives of a union have a member of different type in common, we require a downcast before doing the member access.

6.2 Wildcard, Transitive and Type-based Member-access

The only query form available in object-oriented languages is member access. But that is rather restrictive. To allow for more flexible forms of member access, we provide *wildcard*, *transitive* and *type-based* access. These forms are similar to the concepts of nametest, abbreviated relative location paths and name filters in XPath [14]. However we adapted them to work uniformly on object graphs.

Wildcards allow to access all accessible members of a type without having to know their names. Suppose that we want to have all fields of an `Address`, then we can write:

```
choice{string; int;}* addressfields = Microsoft.*;
```

The wild-card expression returns the content of all accessable fields and properties of the variable `Microsoft` in their declaration order. In this case the stream of strings `"One Microsoft Way"`, `"Redmond"`, `98052`, `"USA"`.

Transitive member-access, written as `e...m`, returns all accessible members m that are transitively reachable from `e` in depth-first order. The following declaration of `authors` (lazily) returns a stream containing all `Author` of all `Book`s in the source stream `books`:

```
Book F = <Book>
           <Title>Faust</Title><Author>Goethe</Author>
         </Book>;
```

```
Book K = <Book>
          <Title>Max Havelaar</Title><Author>Multatuli</Author>
         </Book>;

Book* books = { yield F; yield K; };
string* authors = books...Author;
```

Transitive member access allows to abstract from the concrete representation of a document; as long as the mentioned member is reachable and accessible, its values are returned.

Looking for just a field name is often not sufficient, especially for transitive queries where there might be several reachable members with the same name but of different type. In that case we can add an additional type-test to restrict the matching members. A type-test on T selects only those members whose static type is a subtype of T. For instance, if we are only interested in Microsoft's `POBox` number, and `Zip` code, we can write the transitive query `Microsoft...int::*`.

Note that type based access is also useful for unnamed members, since even if they have no name, they do have a static type.

6.3 Select and Join

The previous sections presented our solutions to querying documents. However for accessing relational data, which we have modelled as streams of tuples, simpler SQL queries are sufficient. Here we only show the integration of the SQL `select-from-where` clause, and defer the discussion of more advanced features such as data manipulation and transactions to a future paper.

The fundamental operations of relational algebra are *selection*, *projection*, *union*, *difference* and *join*. Selection is similar to filter and transforms one stream of tuples into another stream of tuples. Here are two variations of selection:

```
Pokemon* normalPokemons1 =
  select *
  from Pokedex
  where Kind == Normal;
Pokemon* normalPokemons2 =
  select it
  from (Pokemon it in Pokedex)
  where it.Kind == Normal;
```

The first example uses the familiar SQL syntax. Its meaning is provided by the second form, which uses the explicit iterator variable `it` as we have seen before.

We use similar sugar to introduce names for projection. Projection produces a stream of tuples by selecting only certain columns in its input stream:

```
sequence{string Name; Kind Kind;}*
  pokemonAbstract1 =
    select Name, Kind
```

```
    from Pokedex;
sequence{string Name; Kind Kind;}*
  pokemonAbstract2 =
    select new(Name= it.Name, Kind=it.Kind)
    from (Pokemon it in Pokedex);
```

Again, the first declaration shows the traditional SQL syntax, where the second shows the unsugared representation, which explicitly builds the resulting tuple by projecting the required members.

In practice, the result types of SQL queries can be quite involved and hence it becomes painful for programmers to explicitly specify types. Since the compiler already knows the types of sub-expressions, the result types of queries can be inferred automatically. Providing type declarations for method local variables is not necessary, and we can simply write:

```
pokemonAbstract3 = select Name, Kind from Pokedex;
```

without having to declare the type of `pokemonAbstract3`.

Union and difference present no difficulty in our framework. They can easily be handled with existing operations on streams. Union concatenates two streams into a single stream. Difference takes two streams, and returns a new stream that contains all values that appear in the first but not in the second stream.

The real power of `select-from-where` comes from join. Join takes two input streams and creates a third stream whose values are composed by combining members from the two input streams. For example, here is an expression that selects pairs of Pokemeons which have evolved from each other:

```
select p.Name, q.Name
from p in Pokedex, q in Pokedex
where p.Evolved == q
```

Again, we would like to stress the fact that everything fits together. The select expression works on arbitrary streams, whether in memory or on the hard disk; streams simply virtualize data access. Strong typing makes data access secure. But there is no burden for the programmer since the result types of queries are inferred.

7 Conclusion

The language extensions proposed in this paper support both the SQL [4] and the XML schema type system [41] to a large degree, but we have not dealt with all of the SQL features such as (unique) keys, and the more esoteric XSD features such as redefine. Similarly, we already covered much of the expressive power of XPath [14], XQuery [6] and XSLT[13], but we do not support the full set of XPath axis. We are able to deal smoothly with namespaces, attributes, blocking, and facets however. Currently we are investigating whether and which additional features need to be added to our language.

Summarizing, we have shown that it is possible to have both SQL tables and XML documents as first order citizen in an object-oriented language. Only a bridge between the type worlds is needed. Building the bridge is mainly an engineering task. But once it is available, it offers the best of three worlds

8 Acknowledgments

We would like to acknowledge the support, encouragement, and feedback from Mike Barnett, Nick Benton, Don Box, Luca Cardelli, Bill Gates, Steve Lucco, Chris Lucas, Todd Proebstring, Dave Reed, Clemens Szyperksi, and Rostislav Yavorskiy and the hard work of the WebData languages team consisting of William Adams, Joyce Chen, Kirill Gavrylyuk, David Hicks, Steve Lindeman, Chris Lovett, Frank Mantek, Wolfgang Manousek, Neetu Rajpal, Herman Venter, and Matt Warren.

References

[1] The castor project. http://castor.exolab.org/.
[2] The navision x++ programming language.
[3] The Objective-C Programming Language.
[4] G. Bierman and A. Trigoni. Towards a Formal Type System for ODMG OQL. Technical Report 497, University of Cambridge, Computer Laboratory, 2000.
[5] B.Liskov, R.Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. LNCS 114. Springer=Verlag, 1981.
[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, W3C, November 2002.
[7] T. Bray. XML Is Too Hard For Programmers. http://www.tbray.org/ongoing.
[8] P. Buneman and B. Pierce. Union Types for Semistructured Data. In *International Database Programming Languages Workshop*, LNCS 1949, 2000.
[9] L. Cardelli. Types for data-oriented languages. In *EDBT*, 1988.
[10] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *FOSSACS*, 2003.
[11] R. G. G. Cattell, D. K. Barry, R. Catell, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann Publishers, 2000.
[12] A. S. Christensen, A. Muller, and M. I. Schwartzbach. Static Analysis for Dynamic XML. In *PlanX*, 2002.
[13] J. Clark. XSL Transformations (XSLT). Technical report, W3C, November 1999.
[14] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, W3C, November 1999.
[15] R. Connor, D. Lievens, and F. Simeoni. Projector: a partially typed language for querying XML. In *PlanX*, 2002.
[16] M. Fahndrich and R. M. Leino. Declaring and checking Non-Null Types in an Object-Oriented Language. In *OOPSLA*, 2003.
[17] D. C. Fallside. XML Schema Part 0: Primer. Technical report, W3C, May 2001.
[18] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[19] V. Gapeyev and B. Pierce. Regular object types. In *ECOOP*, 2003.
[20] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with High-Level Programming Languages. In *Automated Software Engineering*, LNCS 2028, 2001.
[21] R. Griswold and M. Griswold. *The Icon Programming Language (2nd edition)*. Prentice Hall, 1990.
[22] J. Guy L. Steele. Growing a Language. *Journal of Higher-Order and Symbolic Computation*, 12(3), 1999.
[23] K. Henney. Null Object, Something for Nothing. In *Seventh European Conference on Pattern Languages of Programs*, 2002.
[24] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *International Workshop on the Web and Databases (WebDB)*, 2000.
[25] P. Hudak. Building Domain Specific Embedded Languages. *ACM Computing Surveys*, 28(4), 1996.
[26] M. Kempa and V. Linnemann. On XML Objects. In *PlanX*, 2002.
[27] O. Kiselyov and S. Krishnamurthi. SXSLT: Manipulation Language for XML. In *PADL*, LNCS 2562, 2003.
[28] A. Krall and J. Vitek. On Extending Java. In *JMLC*, 1997.
[29] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. *Lecture Notes in Computer Science*, 1949:297+, 2000.
[30] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *2nd USENIX Conference on Domain-Specific Languages*, 1999.
[31] T.-W. Lin. Java architecture for xml binding (jaxb): A primer, 2002.
[32] B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens. ARGUS Reference Manual. Technical Report MIT/LCS/TR-400, 1987.
[33] E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, 10(1), January 2000.
[34] E. Meijer, D. Leijen, and J. Hook. Client Side Web Scripting with HaskellScript. In *PADL*, 2002.
[35] E. Meijer and M. Shields. XMLambda: a Functional Language for Constructing and Manipulating XML Documents, 1999. http://www.cse.ogi.edu/ mbs.
[36] E. Meijer and D. van Velzen. Haskell Server Pages. In *Haskell Workshop 2000*, 2000.
[37] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
[38] T. Proebstring. Disruptive Language Technologies.
[39] N. Schemenauer, T. Peters, and M. L. Hetland. Simple Generators. PEP-0255.
[40] M. Shields and E. Meijer. Type-indexed Rows. In *POPL*, 2001.
[41] J. Simeon and P. Wadler. The Essence of XML. In *POPL*, 2003.
[42] P. Thiemann. WASH/CGI: Server Side Web Scripting with Sessions and Typed, Compositional Forms. In *Practical Aspects of Declarative Languages*, 2002.
[43] C. van Reeuwijk and H. Sips. Adding tuples to java: a study in lightweight data structures. In *ACM Java Grande/ISCOPE*, 2002.
[44] A. F. Véronique Benzaken, Giuseppe Castagna. CDuce: an XML-centric general-purpose language. In *ICFP*, 2003.
[45] P. Wadler. The Concatenate Vanishes, 1989. http://www.research.avayalabs.com/user/wadler/papers/vanish/vanish.pdf.
[46] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two Little Languages. In *Third Workshop on Scheme and Functional Programming*, 2002.