# Feel Different on the Java Platform: The Star Programming Language

Frank McCabe
Starview, Inc.
fmccabe@starviewinc.com

Michael Sperber
Active Group GmbH
michael.sperber@active-group.de

## Abstract

Star is a functional, multi-paradigm and extensible programming language that runs on the Java platform. Starview Inc developed the language as an integral part of the *Starview Enterprise Platform*, a framework for real-time business applications such as factory scheduling and data analytics. Star borrows from many languages, with obvious heritage from Haskell, ML, and April, but also contains new approaches to some design aspects, such as syntax and syntactic extensibility, actors, and queries. Its texture is quite different from that of other languages on the Java platform. Despite this, the learning curve for Java programmers is surprisingly shallow. The combination of a powerful type system (which includes type inference, constrained polymorphism, and existentials) and syntactic extensibility make the Star well-suited to producing embedded domain-specific languages. This paper gives an overview of the language, and reports on some aspects of its design process, on our experience on using it in industrial projects, and on our experience implementing Star on the Java platform.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages

***General Terms*** Languages, Design, Human Factors

***Keywords*** Multiparadigm Programming, Functional Programming, Compilation, Actors

## 1. Introduction

The origins of Star lie in a Java-based platform developed by Starview that was originally oriented towards *complex event processing* applications [22]. Complex event processing (*CEP*) is a family of techniques for processing very large streams of events, typically in the form of "standing queries" against the events or sets of pattern/action rules. StarRules—as it was known then—was a language that allowed one to express CEP-style rules succinctly. As the Starview Enterprise Platform is an OSGi-based framework that runs on the Java platform, StarRules needed to run on the JVM.

As requirements evolved, CEP became just one of many application areas that the Starview platform can address. The same pressures for generality in the platform lead to a greater interest in extensibility and generality in the Star language.

One particular scenario played out several times: High-level but specific features were forced to evolve to simpler but more general ones. For example, initially Star had a high-level concurrency model to allow multiple agents to process events with some degree of parallelism. This model was elegant and had intuitive semantics, but did not fit all applications that we wanted to support. We refactored the concurrency model with a more general one, which also had the effect of making the concurrency framework lower-level. The replacement of specific but high-level features with more general but lower-level ones occurred several times: for concurrency, relational data structures and the rules formalism itself.

It was partly in response to this that we invested effort in making the Star language extensible via an extensible grammar and macros. In effect, we adopted a domain-specific language methodology for the design of Star itself. Using the extensibility facilities we are often able to present high-level features while basing them on more general more low-level capabilities.

A number of other aspects have influenced Star's design: Successful software projects usually become team efforts. Furthermore, deployment targets will often extend to span a range of devices from smart phones to multi-cluster supercomputers. Finally, integration with other systems if often key to realizing the benefits of a given system.

Another strong influence on Star was the social context: We assumed that any given software project would involve many people. This led us to conclude that features such as clear semantics, safety, strong modularization, and in particular multi-paradigm support are essential rather than being optional.

The foundations for safety in a programming language stem from an expressive type system—the more expressive the type system, the less the temptation to escape from it—and from an easy-to-reason-about semantics. These requirements favor primarily functional languages, especially when it comes to reasoning about parallel and distributed programming.

Star is not a pure language in the sense that it permits programs with assignments and stateful objects. This is both because we wish to permit programmers to build in ways that suits their needs and because some algorithms are easier to express using mutable state. However, the language encourages the programmer to program more declaratively. Star enables procedural programming whilst constraining features to avoid some pitfalls of global hidden state.

***Contributions*** Star fits into a unique niche in the crowded landscape of languages for the JVM. Star's contributions to this landscape are the following:

- Star is a unique and successful combination of paradigms and concepts from a variety of languages, many of them from outside the Java platform. For each concept, we tried to choose the



*PPPJ'13*, September 11–13, 2013, Stuttgart, Germany.

http://dx.doi.org/10.1145/2500828.2500837

best technology available that was both proven and fit well with the overall language.

- Star provides a fresh approach to implementing embedded domain-specific languages via extensible syntax and macros.
- Star provides a fresh look at the language infrastructure needed to implement agents via a taxonomy of *speech actions*.
- Compiling Star to the JVM runs up against JVM restrictions and omissions: We describe our solutions to the problem of representing arbitrary-arity functions, and the JVM's method-size restriction.
- Even though Star feels very different from Java, the learning curve for Java programmers is smooth.

While we expect the Star implementation to become open source later in 2013, it is available now to interested parties upon request.

## 2. Language synopsis

Star is a *statically typed*, *functional*, and *strict* language.

- *Statically typed* means that Star, like Java, has a type system that ensures at compile time that operations cannot be applied to objects of invalid type at run time. Star's type system is quite different from Java's however, and is much closer to the Haskell type system in spirit. In particular, it supports type inference, true parametric polymorphism, constrained polymorphism with *contracts* (Star's version of Haskell's type classes), functional dependencies, and existential types.
- *Functional* means that, in Star, functions are first-class values. With the advent of Project Lambda in Java, this is less of a distinction than it used to be. However, using functions in Star is—due to native support in the type system and type inference—much less painful than it is in Java 8, comparable to Scala or Clojure.
- *Strict* means that Star's evaluation model is like Java's: When evaluating a function call, a Star program first fully evaluates its arguments before passing them and calling the function. This distinguishes it from Haskell's *lazy evaluation*.

Star is not *purely functional* like Haskell: it supports first-class references, like ML. However, Star has, like Haskell and F#, a monadic framework for describing effectful *computations*. (See Section 4.11.) This is particularly useful for expressing concurrency and making efficient use of parallelism.

Star is very different from Java: It is not primarily object-oriented, favors (purely) functional programming, and its syntax has a feel quite different from that of Java.

Star is also quite different from most other languages on the Java platform. The currently popular choices—Scala, Clojure, Kotlin, Groovy—all are explicitly languages for programming the Java platform, and the platform "shines through" in day-to-day programming. In contrast, even though Star does offer Java interoperability, Star's design is significantly more independent from the constraints of the host platform.

Many of the language elements in Star—including the operator precedence grammar, much of the macro processing framework and the query expressions—were explored in an earlier multi-agent language called April [23]. Star, however, greatly extends the type system of April. Where April had processes—which are similar to Erlang processes—Star has actors and speech actions.

## 3. Texture

This section tries to convey a sense of what it feels like to program in Star by describing aspects of its texture, things that are most immediately apparent when using it, a few simple examples. Here is a simple Star function definition with (hopefully) obvious semantics:

```
doubleMe(x) is x + x;
```

Here is another function calling the just-defined `doubleMe`:

```
doubleUs(x, y) is doubleMe(x) + doubleMe(y)
```

The following function uses a conditional to double numbers less-or-equal than 100, and leave all other numbers alone:

```
doubleSmallNumber(x) is (x > 100) ? x | x*2;
```

Function definitions can use multiple clauses, pattern matching, and guards:

```
lucky(7) is "LUCKY NUMBER SEVEN!";
lucky(x) default is "Out of luck, pal!";

factorial(0) is 1
factorial(n) where n > 0 is n * factorial(n-1);
```

Like most functional languages, Star supports "cons lists" natively. Cons lists are singly-linked lists. Thus, there are two kinds of cons lists: the empty cons list `nil`, and `cons(`*`x, xs`*`)` where *x* is the first element, and *xs* is the rest of the list. For example, the cons list:

```
cons(4, cons(8, cons(15, cons(16, cons(42, nil)))))
```

has the numbers 4, 8, 15, 16 and 42 in it. Star supports an alternate, more readable notation for `cons` lists too:

```
lostNumber is cons of { 4; 8; 15; 16; 42 }
```

Programmers can define functions over cons lists via pattern matching:

```
listSum(nil) is 0;
listSum(cons(x, xs)) is x + listSum(xs);
```

The first clause is for lists with no elements. The second clause computes the sum of the rest of the list via a recursive call, and adds the first element to it.

Star supports higher-order functions. The following `filter` function, familiar from other functional languages, selects those elements of a list that match a given predicate:

```
filter(p, nil) is nil;
filter(p, cons(x, xs)) where p(x) is
  cons(x, filter(p, xs));
filter(p, cons(x, xs)) default is filter(p, xs);
```

Except in a few special situations Star does not require that programmers annotate programs with types. Star has a sophisticated type system that supports automatic *type inference*, which is why the above examples show no explicit types. However, a programmer can supply type annotations, which often improve readability. Here is a type annotation for `filter`:

```
filter has type for all t such that
  ((t) => boolean, cons of t) => cons of t;
```

In the type, `t` is a *type variable* that says that `filter` is polymorphic and works over list elements of arbitrary types. The function accepts a function from `t` to `boolean` and a list of type `cons of t`, i.e. a cons list with elements of type `t`. The `filter` function returns a filtered cons list of `t`s.

Type annotations are not mandatory: As in other functional languages based on Hindley/Milner-typing, functions are values, and thus a naive approach to mandatory type annotations would require them on every declaration. This, however, would make the code quite verbose, and the cognitive cost of the resulting clutter is not worth the benefit, as the experience with Java shows. For

example, types are often obvious in local and helper declarations. Therefore, type annotations are always optional in Star.

## 4. Language overview

This section gives a semi-systematic overview of the Star language, grouped by language aspect.

### 4.1 Syntax

Star's syntax mostly eschews keywords that consist of special characters, instead it relies on words to convey meaning. This makes the language somewhat verbose in characters, but also tends to nudge developers to make the code self-explanatory. This is in contrast to, for example, Haskell, where special-character operators often make programs concise at the cost of understandability for outside readers.

Unlike many other languages, Star does not have a fixed grammar. Instead, its syntax is specified via an *operator-precedence grammar* [26]. This allows using functions as infix operators. Also, it makes the syntax extensible by new syntactic constructs defined via *macros* (see Section 4.13), without forcing the language to have S-expression syntax.

### 4.2 Data types

Star has a rich language for defining compound data types building on the heritage of ML and Haskell, with tuples, records, algebraic data types, and type constructors:

***Tuples*** The simplest way to combine several values into a compound is via a *tuple*. For example, the tuple

```
(1971, "Mike Sperber", true)
```

is a tuple consisting of three values, with this type:

```
(integer, string, boolean)
```

In addition to providing "lightweight compound objects," programs often use tuples to return multiple values from a function. Note that, unlike Haskell or ML, Star has an intrinsic notion of a multi-argument function and does not rely on tuples or currying to provide them.

***Records*** Records are compound values with named fields. Here is an example:

```
type person is someone{
  name has type string;
  birthday has type date;
};
```

Star allows deconstructing record values both via pattern matching or the familar dot notation ("`p.birthday`").

***Algebraic data types*** *Algebraic data types* allow expressing the notion of *sum types*, where a value of the type can be one of several kinds. Here is an example:

```
type accountTransaction is
     payment(integer, string)
  or withdrawal(integer, string);
```

This means what the wording suggests: An account transaction is either a payment or a withdrawal, either of which carries an integer (presumably the amount) and a string (presumably a description). Functions can easily deconstruct values of such types via multiple clauses and pattern matching:

```
value(payment(amount, desc)) is amount;
value(withdrawal(amount, desc)) is -amount;
```

Algebraic datatypes combine seamlessly with records:

```
type entity is
  somebody{
    name has type string;
    birthday has type date;
  }
  or something(integer);
```

In the degenerate case, algebraic datatypes can express enumerations:

```
type quality is good or bad;
```

***Type constructors*** Star supports type constructors:

```
type tree of t is
     leaf
  or node(t, tree of t, tree of t);
```

Here, `tree` is a type constructor for binary trees that takes a type parameter `t` for the type of the node labels.

***Sequences*** Cons lists are defined with this type definition:

```
type cons of t is
     nil
  or cons(t, cons of t)
```

The cons-list literal

```
cons of { 4; 8; 15; 16; 23 };
```

is equivalent to this expression:

```
cons(4,cons(8,cons(15,cons(16,cons(23,nil)))))
```

The *sequence notation* `cons of` for cons lists is more convenient, and generalizes to arbitrary data types. For example, the sequence notation can also apply to the `tree` data type. The clumsy notation:

```
node(2, node(1, leaf, leaf), node(3, leaf, leaf))
```

could be written as follows, imposing an arbitrary order on the tree elements:

```
tree of {1; 2; 3}
```

The sequence notation means that even collections with complex internal structure can be written in a natural way by the client programmer. This flexibility of notation is achieved by means of a partnership of macros and contracts. See Section 4.6.

### 4.3 Theta environments

Star supports a general notion of local definition through *theta environments*[1]: A theta environment provides bindings for an expression or binding, and can contain definitions for values, types, contracts, as well as imports and type annotations. Bindings can be mututally recursive. Here is an example:

```
{
  x has type integer;
  x is 5;
  type t is foo(integer) or bar(boolean);
  f(x) is x + 1;
}
```

Theta environments can occur in a variety of contexts: `let` expressions allow using the bindings from a theta environment in an expression:

```
let {
  area is radius * 2 * pi;
} in area * height;
```

[1] The term "theta environment" goes back to Star's logic-programming heritage, where formal semantics are often written using $\theta$ for variable-binding environments.

Also, a program can use a restricted form of theta environment to construct a record value:

```
type accountState is accountState{
  balance has type () => integer;
  deposit has type (integer, string) => accountState;
  withdraw has type (integer, string) => accountState;
  selectTransactions has type
    ((accountTransaction) => boolean) =>
      cons of accountTransaction;
};

makeAccountState(currentBalance, transactions) is
 accountState{
  balance() is currentBalance;
  deposit(amount, desc) is
     makeAccountState(currentBalance + amount,
                      cons(payment(amount, desc),
                           transactions));
  withdraw(amount, desc) is
     makeAccountState(currentBalance - amount,
                      cons(withdrawal(amount, desc),
                           transactions));
  selectTransactions(pred) is
    filter(pred, transactions);
 };
```

### 4.4 Functions

Like any other functional language, Star supports first-class functions. Function expressions use the `function` keyword. Here is an example:

```
filter((function (x) is x%2=1),
       cons of { 2; 3; 5; 7 })
```

### 4.5 Pattern matching

Section 3 already showed some examples for pattern matching. Star's patterns include advanced features such as disjunction, regular expressions, guards, type casts and tests, and non-linear patterns.

Unlike most other functional languages with pattern matching, Star does not guarantee left-to-right matching of the function-definition clauses. A "fall-through case" has to be marked with the `default` keyword. Here is an example:

```
isEven(n) where n%2=0 is true;
isEven(_) default is false;
```

This encourages the programmer to specify pattern-matching definitions more declaratively than in languages like ML or Haskell.

### 4.6 Constraints and contracts

Recall the definition of the `doubleMe` function in Section 3:

```
doubleMe(x) is x + x;
```

Possible types for `doubleMe` would be `(integer) => integer` or `(float) => float`. In principle, `doubleMe` could work on *any* type as long as it supports addition. Star, similarly to Haskell, allows generalizing `doubleMe` over both:

```
doubleMe has type for all t such that
  (t) => t where arithmetic over t;
```

This type means that `doubleMe` works for all types `t` that support arithmetic, as expressed by the *contract constraint* `arithmetic over t`: A *contract* is similar to a Haskell type class: A contract holds for a type if a set of pre-declared functions are available for that type.[2] In this case, the `+` function is part of the `arithmetic` contract, and therefore incurs the `arithmetic` constraint in the type of `doubleMe`.

Programs can define *implementations* of contracts by providing function definitions. For example, the `equality` contract is defined like this:

```
contract equality over t is {
  (=) has type (t,t)=>boolean;
};
```

Here is an implementation of `equality` on a type for "users" that have a unique `id` field. The example assumes that `integerEquals` is a primitive function for comparing integers:

```
type user is user{
  id has type integer;
  name has type string;
  address has type string;
};

implementation equality over user is {
  u1 = u2 is integerEquals(u1.id, u2.id);
};
```

Contracts can be used for some of the same tasks as interfaces and abstract classes in OO languages. Moreover, contracts allow defining implementations "after the fact", separate from the definition of type, and can feature the underlying type in any position, not just the receiver of a method call. As compared with Java's `equals` method, the `equality` contract allows restricting the use of equality to types where the operation actually makes sense.

Programmers can also declare contracts over type constructors. This is needed to define generic versions of advanced concepts from functional programming, such as *monads* [37] or *arrows* [16].

Star supports some generalizations of contracts in the spirit of Mark Jones's theory of qualified types [18], namely multi-parameter contracts and functional dependencies [19]. It also supports constraints that are not contract constraints: In particular, it allows constraining types to record types that have certain fields.

### 4.7 References

Star does not support directly reassignable variables like Java or Scala, but does provide first-class *reference cells* in the spirit of ML. Cells representing mutable variables or fields of underlying type *t* have type `ref t`.

A reference cell is simply an object with a single member: its value. In ML, using reference cells is awkward as a program always needs to explicitly dereference them, using the `!c` notation for dereferencing cell *c*). Recognizing that this awkwardness would make Star less palatable for programmers coming from Java, Star automatically inserts cell dereference operations where needed. Here is an example:

```
type account is account{
  state has type ref accountState;
};
makeAccount(balance, transactions) is
 account{
  state := makeAccountState(balance, transactions)
 };

deposit has type (account, integer, string) => ();
deposit(acc, amount, desc) do
  acc.state := acc.state.deposit(amount, desc);
```

[2] Star's version of type classes differs from Haskell not in concept, but in the specific set of features, and subtle aspects of its semantics.

Note that filling the `state` field in `makeAccount` does not require explicitly constructing a cell, and the right-hand-side occurrence `acc.state` in `deposit` does not require an explicit dereference.

In the few subtle cases where ambiguity prevents Star from correctly divining where dereferences are needed, the `ref` keyword (at the expression level) turns the inference off.

### 4.8 `null` safety

Null references are a notorious problem in Java programs. (And not just in Java programs [15].) Hence, Star, like most functional languages, eschews `null` entirely. Instead, Star invites programmers to use the option type common in functional programming for results that may be a value or "absent":

```
type option of a is none or some(a);
```

### 4.9 Relations and queries

Star has a range of collection types besides cons lists. A *relation* is similar to the SQL concept of a table. Programmers can write literal tables using the sequence notation:

```
t is relation of {
  (1971, "Mike Sperber", true);
  (1926, "Elizabeth Windsor", false);
  (1953, "Frank McCabe", false);
  (1949, "Bruce Springsteen", true)
};
```

Star allows the programmer to formulate relational queries, generalizing the list comprehensions familiar from other functional languages:

```
cons of {
  all Who where
    (Year, Who, _) in t
    and Year>=1946 and Year<=1964
    order by Year
}
```

This expression yields the expected result:

```
cons of {
  "Bruce Springsteen";
  "Frank McCabe" }
```

### 4.10 Satisfaction semantics of conditions

Conditionals and guards in Star work as in other functional languages for conditions that are straightforward boolean expressions:

```
(age > 18) ? "old" | "young"
```

However, conditions may also include *unbound* variables, in which case they act as constraints on the unbound variables, as in `where` clauses to queries. Evaluation tries to *satisfy* such constraint conditions with appropriate bindings. Here is an example:

```
X in male and ("fred",X) in parent
```

This looks for a value for `X` in relation `male` that has a corresponding tuple in relation `parent`, and, if used in a conditional, binds `X` to that value in the consequent:

```
(X in male and ("fred",X) in parent) ? X | "unknown"
```

The satisfaction semantics enables elegant expression for many "query conditions" and lets the programmer avoid awkward binding constructs.

### 4.11 Computation expressions

*Monads* are a convenient form to describe computations as first-class objects [37] and enjoy widespread support in the functional-programming community. Informally, a monad is a type constructor `M` that can be applied to a type `t`, so that `M of t` is the type of monadic computations of result type `t`: A monadic computation is roughly an object that computes a value or values of type `t`. A monad comes with two operations commonly called `unit` and `bind` of the following types

```
unit has type for all t such that
  (t) => M of t;
bind has type for all t,u such that
  (M of t, (t) => M of u) => M of u;
```

The `unit` function allows, for a value, to produce a computation that produces exactly that value, and `bind` allows composing a computation with a function that looks at its value and produces a new computation, based on that value. Monads are a tremendously useful organizational tool for functional programs. A full treatment is, unfortunately, outside the scope of this paper. We refer the interested reader the extensive literature on the subject.

Writing monadic expressions using explicit calls to a monad's `bind` operation is awkward. Even Haskell's and F#'s syntax for computation expressions forces programmers to linearize their programs, which is still occasionally awkward. Star takes a slightly different syntactic approach to monadic expressions. Here is a simple example using a result type for a simple expression interpreter:

```
type expr is
     constant(integer)
  or binary(binop, expr, expr);

type binop is add or sub or mul or div or mod;

type result of t is val(t);
```

This type can be made into a monad. A monadic computation expression can use this type and has the following form:

```
result computation { ... }
```

The code in braces is similar to a theta environment, but can mention `valis` for the monadic `unit` and `valof` to bind the value of a computation.

```
type binfct is alias of
 ((result of integer, result of integer)
    => result of integer);

lookupFtab has type (binop) => result of binfct;

eval has type (expr) => result of integer;
eval(constant(i)) is
  result computation {
    valis i;
  };
eval(binary(op, l, r)) is
  result computation {
    res is (valof lookupFtab(op))(eval(l), eval(r));
    valis res;
  };
```

Note that `valof` can appear anywhere, not just in a direct binding form, as in Haskell's `<-` or F#'s `let!`. The Star compiler performs the transformation to monadic form automatically.

### 4.12 Concurrency and parallelism

High-performance computation requires making effective use of multicore architectures. We eschewed event- and callback-driven systems [36], and instead considered numerous high-level programming paradigms for multithreaded programming. We found two of them particularly attractive because of their generality and composability, and have implemented them in Star:

**Monadic combinators** (for a monad called *task*) for expressing asynchronous and parallel computations, and combining their results [35].

**Concurrent ML** (CML) for orchestrating complex choreographies in concurrent programs [29].

In particular, CML won out over the join calculus [11], which is similarly expressive but—we felt—not sufficiently proven in practical applications yet. We considered Scala's approach of constructing event-driven programs via a combinator library [13], but consider programming with our monadic combinators more straightforward and convenient. While there is some overlap between the monadic combinators and CML's combinators, providing both paradigms in a single language provides some useful synergies: The Concurrent ML substrate is implemented on top of the monadic substrate. Established techniques for implementing CML on parallel platforms [28] translate smoothly.

Here is a simple example for a stream filter using the CML abstractions:

```
streamFilter(P, inCh) is
  let {
    outCh is channel();

    loop() is task {
      I is valof (wait for incoming inCh);
      if P(I) then
        perform send(outCh,I);
      perform loop();
    };
    { ignore background loop() }
  } in outCh;
```

The `streamFilter` function accepts a predicate function and an input channel, and returns an output channel: Its job is to accept messages on the input channel, and forward those messages for which the predicate returns true to the output channel.

The `channel()` call allocates a synchronous communication channel called `outCh`. The filter runs as a background task implemented in the `loop` function. This function waits for an incoming message on `inCh`. It then checks the predicate, and conditionally uses `send` to forward the message to `outCh`. (The `perform` construct is shorthand for a valof that ignores the value of the expression.) It then loops. The `background` construct starts the process.

To our knowledge, the combination of monadic combinators and Concurrent ML is quite unique. In particular, we are not aware of any other monadic implementations of concurrency using a trampoline on the JVM.

### 4.13 Macros

First-class functions go a long way towards enabling the programmer to express her thoughts in an idiomatic manner. Some instances of abstractions as well as the construction of true domain-specific languages require syntactic abstraction, however. Star allows the definition of syntactic abstractions as *macros*.

For example, a type definition of *promises* might be as follows

```
type promise of t is promise(() => t);
```

A promise is a "delayed value" that is not computed upon creation, but instead only when needed. The promise represents this value by a nullary function that the program calls or "forces" when it needs the value:

```
force(promise(mf)) is mf();
```

Creating a promise therefore involves creating a function, typically with a *function expression*:

```
promise((function () is f() + g()));
```

As the `function` tends to appear many times in any program that uses promises, it is convenient to abstract it into a new syntactic form using a macro:

```
#delay(?exp) ==> promise((function () is ?exp);
```

This definition states that `delay(f()+g())` is equivalent to the above expression.

The combination of the extensible grammar, infix notation, and macros, gives great flexibility to the syntax. For example, instead of writing:

```
filter(isEven, cons of {3;4;5})
```

(where `isEven` is a function that returns `true` on even numbers, `false` on others) we might prefer to write:

```
cons of {3;4;5} >> isEven
```

We can do this by defining `>>` as a new operator:

```
#infix(">>",900)
```

and defining a macro that maps a `>>` expression into an `filter` application:

```
# ?S >> ?F ==> filter(F, S)
```

Arguably, this macro—with infix syntax and a name that consists of special characters—is exactly what we complained about in Section 4.1. Indeed, macros can obfuscate programs greatly. However, when properly used, they can serve to enhance readability by removing syntactic clutter and introducing notation closer to human language. In particular, macros are powerful tools for introducing domain-specific notation into embedded domain-specific languages.

The macro language can expand more complex syntactic patterns as well:

```
# #(select all from ?P in ?S)# ==>
    cons of { for P in S do elemis P }
```

Thus, Star's macros are similar to Scheme's `syntax-rules` [7] in spirit, but are not hygienic.

The macro language is powerful enough to express complex programs at the macro level. See Section 6.1.

Star's approach to syntax and macros has some similarities with Honu [27]: Honu's parsing engine is also based on operator precedence, and Honu, like Star, allows defining macros via pattern matching over syntactic categories. In both Star and Honu, the set of syntactic categories is extensible.

Unlike Honu, which interleaves parsing and macro expansion, Star implements a strict pass separation between the two. This disallows macros expanding into macro definitions or operator declarations, but simplifies the syntactic model greatly.

### 4.14 Pattern abstractions

In Star, pattern abstractions are first-class values. In particular, they can be used to declare *views* on algebraic data types such as in this example for lazy sequences, represented by *even streams* [38]. The example refers to the `promise` type from the previous section:

```
type evenStreamNode of a is
     EvenNilC
  or EvenConsC(a, evenStream of a);
type evenStream of a is alias of
  promise of evenStreamNode of a;

evenConsP(x, xs) from
  (p where force(p) matches EvenConsC(x, xs));
```

Now `evenConsP(x, y)` is a pattern that matches a non-empty even stream, forcing it in the process.

### 4.15 Actors

Agenthood is often a useful metaphor in structuring large programs [39]. The intuition is that an agent represents a focus of responsibility within a system: by structuring the system in terms of who is responsible for what it can make large systems more tractable.

Agents are typically designed to be 'in charge of' their own area of responsibility and agents collaborate by 'talking' to each other. The messages exchanged between agents are not just data: Agents also communicate intentionality and distribute responsibility by asking other agents to 'do stuff for them.' This amounts to the power of speech.

The reason that agents are an effective metaphor is that it is easier to design an entity that has limited responsibility and concern as well as speech. Furthermore, in a distributed system where agents reside on different machines, some degree of self-responsibility is necessary. Agenthood is also useful when the application programmer is tasked with modeling aspects of the world that are self-actuated: for example when modeling the behavior of people or of machines.

Star supports the implementation of agents with two key concepts: *speech actions* and *actors*.

Speech actions were first investigated by Austin [3] in the 1940's as a vehicle for understanding the role of speech in human society. Since that time the basic ideas have been progressively formalized by Searle [33] and standardized in KQML [8] and FIPA [9].

Within Star, a speech action can be viewed as a generalization of a method call where the method to be invoked can be a complete script or expression. Star supports three performatives: `notify`—which corresponds to one entity *informing* another that something has happened—`query`—which corresponds to a *question*—and `request`—which corresponds to a request to perform an *action*.

A `notify` is written:

```
notify acc1
  with payment(1300, "paycheck")
  on transactions;
```

This `notify` means that the `transactions` channel should handle `payment(1300, "paycheck")`, or, in the terminology of speech act theory:

> Inform agent `acc1` that `payment(1300, "paycheck")` has occurred.

The `notify` speech action does not explicitly refer to time. This is consistent with the architectural principle of separation of concerns given that there may be multiple senses of time: the time of the occurrence, the time of its being noticed, or the time of this speech.

Here is an actor[3] that can accept these `notify`s:

```
makeAccountActor() is actor {
  var state := makeAccountState(0, nil);

  on payment(amount, desc) on transactions do
    state := state.deposit(amount, desc);
  on withdrawal(amount, desc) on transactions do
    state := state.withdraw(amount, desc);
};
```

[3] Star's actors should not be confused with what is arguably the original definition of actor by Hewitt [1]. Hewitt actors are a representation of concurrent programs; Star actors may or may not be concurrent.

The `state` variable holds the internal state of the actor. The two *event* rules `on ... on` define how the actor reacts to messages on the `transactions` channel.

The actor can also contain function definitions:

```
makeAccountActor() is actor {
  ...
  balance() is state.balance();
  selectTransactions(pred) is
    state.selectTransactions(pred);
};
```

A program can use these with the second kind of speech action—the `query`. For example, to query an (augmented) account actor for its balances one might use:

```
query acc1 with balance()
```

There is no special kind of rule within an actor that is used to respond to `query` speech actions. Instead the response to the query is determined simply by evaluating the whole expression relative to the actor's theta environment. Programmers usually assume that query speech actions do not modify the state of the listener.

The final form of speech action `request` assumes that the listener should *do* something. For example, we can ask an actor to clear the balance and transaction history:

```
request acc1 to
  { state := makeAccountState(0, nil) }
```

Notice that the argument of a `request` is an action. It is possible for the listener to the `request` to decline to perform this request. This ability (or lack of) to not react to speech actions is a characteristic of the responding actor.

The Star `actor` represents the simplest possible entity that can respond to speech actions. In that sense, an `actor` is the simplest possible mechanism for embodying responsibility. Other entities may also receive speech actions: they only need to implement a special `speech` contract.

There are other entities that also implement the `speech` contract. In particular, *concurrent actors* are a variation on the basic actor that performs its actions on a separately executing task; see Section 4.12. Moreover, the Starview Enterprise Platform expresses a "model" for processing events in terms of *ports*, which are a variant of actors which may be wired up separately from their construction with *port adapters*. Port adapters allow queries to be forwarded, to be split into sub-queries to be answered by different components and allow speech actions to be translated on-the-fly to facilitate the necessary 'gluing' together of components written by different people. Moreover, the platform special features adapters that translate between internally generated speech actions and external services such as databases or web services.

An agent could be seen as being the simplest entity that both responds to speech actions and has some awareness of its own goals and activities. Thus actors and agents span a range of scales from the very small to the very large: but with a unified representation of 'units of collaboration': the speech action. In particular, Star actors are able to operate on a reified representation of `query` and `request` speech actions, similar to (but developed independently from) LINQ's approach [24].

### 4.16 Java interoperability

Given that Star operates on the JVM platform, a certain amount of interoperability between Star and Java is essentially free. Unfortunately, the differences in semantics between Star and Java are sufficiently deep that *transparent* interoperability is effectively impossible. However, Star does permit Star programs to access libraries

written in Java and conversely, a Java program may access functions and data types written in Star.

In practice, interoperability of data is limited to so-called container classes and interoperability of functions is limited to statically defined Java functions.

## 5. Platform issues

As Star is so different from Java, and the JVM is primarily designed to support Java, compiling Star to the JVM meets some significant impedance mismatches and difficulties. This section gives an overview of the issues.

### 5.1 Representation of functions

The JVM does not directly permit a function to exist on its own: There are only methods that must be associated with classes. Nor do Java methods have an intrinsic concept of free variables.

Thus, this simple Star program:

```
K(X) is (function(Y) is X)
```

is not directly possible on the JVM because the `K` function returns a function, which is not a first class value in Java.

We use a fairly common technique for representing Star functions: each function is represented as a class with a method called `enter` to represent the function.

Borth `K` and the anonymous function embedded within it are represented as classes. The following Java code paraphrases the JVM code that the Star compiler generates:

```
class K456 implements function1_tV_function1_tV_tV{
  IValue enter(IFunction X){
    return new F34568(X);
  }
}
class F345 implements function1_tV_tV{
  IFunction X;

  F345(IFunction X){
    this.X=X;
  }
  enter(IValue Y){
    return X;
  }
}
```

The free variable `X` that is part of the anonymous function is represented internally as an instance variable of the `F345` class.

The compiler use a munged name `K456` to denote the class that implements the `K` function because Java's name scoping rules do not map directly to Star's scoping rules. Furthermore, class files contain references to other classes as specially formatted strings so we need to treat Star identifiers with some care on the JVM.

The `function1_tV_tV` and `function1_tV_function1_tV_tV` interfaces are synthesized to represent the erased Java types that Star programs process. The `IValue` interface denotes any valid Star value and the `IFunction` interface denotes any Star function.

The Java type signature for the `enter` method tries to capture as much as possible of the original Star types as possible. In particular, if a function is defined to take `int` arguments, then its signature (of a function `plus` that accepts and produces integers) will reflect this:

```
class Plus456 implements function2_int_int_int {
  int enter(int X,int Y){
    return X+Y
  }
}
```

However, even though Java 5 and onwards have some concept of type variables, the JVM's understanding of types remains at Java 1 levels. This means that some combinations are difficult to arrange for. For example, the following expression is legal Star:

```
K(plus)
```

However, the signature for plus does not match the permitted signature for the argument of the `K456` class. This expression, which is rendered:

```
K456 K = new K456();
Plus456 P = new Plus456();
K.enter(P);
```

is not legal Java because of the differences in signatures. To ease this problem, the Star compiler constructs a cascade of interfaces for each function class: each is more general until *something* fits.

One of the unfortunate side-effects of this mismatch is that the Star compiler must generate many extraneous type cast instructions and must generate additional copies of the `enter` method for different signatures:

```
class K456 implements function1_tV_function1_tV_tV,
  function1_tV_tV,
  IFunction ...
class Plus456 implements function2_int_int_int,
  function_2_tV_tV_tV,
  IFunction ...
```

The Star classloader uses a special classloader that ensures that, even though multiple identical synthesized interfaces may be generated, only one copy of each synthesized interface code is actually loaded.

In addition to the specialized signatures, the Star compiler constructs a completely generic entry point that can handle any legal Star sequence of values:

```
class K456 implements ... {
  IValue enter(IFunction X) { ... }
  IValue enter(IValue ... args){
    return enter((IFunction)args[0]);
  }
}
```

In most situations the compiler can avoid invoking this generic entry point; but it is still used more often than is desirable. Consequently, the semantic mismatch between Star and Java shows up in a large amount of fundamentally extraneous code that must be generated to the JVM's internal verifier.

### 5.2 Proper tail calls

A staple complaint of functional programmers on the JVM is the lack of proper tail calls [30], which hurts Star programmers in several ways:

- Programmers have to resort to using `while` loops and imperative programming to implement iterative processes.
- While the implementation of `notify` speech actions via method invocation is fast, it also does not support unbounded transfer of messages. This requires special care on the part of the programmer when using `notify`.

Unfortunately, no language on the JVM has found an satisfactory solution to this problem: Most, like Star, offer specialized looping constructs. Scala offers local tail calls via an annotation. We may implement local tail recursion to alleviate the worst of the problem, but do not expect a general solution until the Java platform supports proper tail calls natively, something that has been under consideration since at least 1996 [34].

### 5.3 JVM-level threads

The sophisticated programming techniques supported by Star for concurrent and parallel programs (see Section 4.12) rely on an unbounded and large number of threads. Unfortunately, the JVM imposes significant penalties on programs that use many JVM-level threads [12]: Each threads takes up a significant amount of storage, creating a new thread is expensive, the total number of threads that a program can create is bounded, and, perhaps most seriously, a JVM thread cannot be garbage collected unless its execution has terminated.

Unfortunately, a typical CML programs allocates many threads, and thus essentially assumes that threads are cheap to create, and that a single program can allocate an unlimited number of them. This precludes using JVM threads directly as CML threads.

Fortunately, the monadic approach for concurrency and parallelism helps solve this problem: Star uses its own, cheap thread representation. The system easily scales to 100.000s of "green threads".

The Star compiler transforms computation expressions into monadic form [4]. (See Section 4.12.) This transformation of task-computation expressions to monadic form makes the continuations of synchronizing calls explicit, and turns them into tail calls. This enables Star to reify the context of an asynchronous continuation. In particular, when a synchronizing computation needs to block, it *returns immediately* instead of blocking the underlying JVM thread. A Star scheduler provides a trampoline to multiplex several computations onto a single JVM thread, and makes the context switch between them extremely fast. This technique borrows from F#'s implementation of the `Async` monad [35]. A Star program, upon startup, only allocates JVM-level threads to the extent needed to exploit parallelism, and one of its own schedulers on each of these JVM threads.

The monadic transformation comes at a cost, as it converts linear code into a chain of function calls, which slows down execution. However, this cost is local to computation expression, and mainly occurs at blocking calls, where the program would need to suspend or block anyway. Still, this technique effectively makes the program perform tasks that properly belong in the domain of the VM itself, and that the VM could address more efficiently.

### 5.4 Size restrictions

Compilation to the JVM is complicated by further restrictions:

1. The JVM only supports methods up to 64k bytes of code.
2. A JVM method can only have up to 255 parameters.

The restrictions are acceptable for most directly hand-written methods. (There are, however, numerous real-world examples in various JVM languages that had be to be rewritten to meet these restrictions.) However, practical uses of Star's macros effectively produce *machine-generated code* that exceeds those limits quite often.

The method-size limit is the more immediately painful restriction.[4] We chose to mediate the problem by splitting large methods into multiple smaller ones. The necessary transformation would be easier to perform at the source-code level, but the compiler lacks information about the size of the generated byte code at that stage, and keeping size-estimate code in synch with the actual code generation poses maintenance problems we were not prepared to handle.

Instead, we split methods at the byte-code level using a general extension of the ASM [6] byte-code generation engine:[5] When ASM detects that the generated code for a method is too large, it invokes the splitter, which splits it into several methods.

To that end, the splitter constructs a flowgraph, computes cycle equivalence on the graph [17], and uses it to break out sub-trees of the tree of strongly-connected components of the flowgraph that have no backedge going over them; it then moves these sub-trees into separate methods and generates calls to them.

The generated call transfers the current local-variable frame and the stack over to the split-out method. These can get large and frequently hit the *other* restriction of only 255 parameters for a method. (In rare cases, we have also hit the restriction on the signature size.) This in turn has forced us to implement a liveness analysis on the frame to cut down on the size of parameters passed.

The process just described is effective for most Star programs that run into the method-size restriction. Still, the splitter fails on programs that have large loop bodies, as the back edges would necessitate transferring control back and forth between the split-out methods. This in turn is not possible on the JVM because of the lack of proper tail calls. Using a trampoline for the control would be prohibitely expensive, as the program would have to allocate heap objects to hold the frame contents.

To successfully handle a larger set of programs, we plan to extend the splitter to split out methods "in the middle" that correspond to subexpressions in the original program. This will require a more sophisticated data-flow analysis an analysis of the flowgraph.

## 6. Experience

This section highlights two applications developed Star. One is an embedded DSL for transforming database transaction, the other is an application framework for semiconductor-fab scheduling. An additional subsection describes our experience with Java programmers transitioning to Star.

### 6.1 Transforming database transactions

One of the most common yet most painful exercises in programming applications is transforming data from one format to another. This is especially so in the context of database programming because the natural data structures present in databases are so different to the natural structures available in programming languages.

For example, consider the humble invoice. Abstractly, one can consider an invoice to be a pair consisting of information about the invoice itself, who the customer is, the date of the transaction, and a collection of line items. Each line item details the item purchased, its quantity and price. Here is a corresponding Star type:

```
type invoice is invoice{
  customer has type Customer;
  invoiceDate has type date;
  items has type cons of lineItem;
};
type lineItem has type lineItem{
  SKU has type skuType;
  quantity has type integer;
  price has type float;
};
```

The problem is that databases typically cannot represent invoice data in this way because a tuple in a table may not contain another table, only a reference to it. On the other hand, databases are often the preferred method for storing invoices.

Star's query sub-language—which has some similarities to LINQ—allows us to express the essence of the mapping between a relational representation of invoice data and a `invoice` representation as a query expression. For example, a list of recent invoices may be constructed using the query:

[4] It is also the most frustrating restriction, as the JVM class-file format could accomodate larger methods with very little modification.

[5] The extension is available at `http://bitbucket.org/sperber/asm-method-size`. It is completely transparent, and could work for any compiler using ASM to generate JVM byte code.

```
cons of {
  invoice{
    customer = C; invoiceDate=D;
    items = cons of {
      all { lineItem{ SKU = S; quantity=Q; price = P}
        where dbItem{ SKU=S; quant=Q; price=P;
                      invNo=invId} in dbItems} }
  } where dbInvoice{ invNo=invId;
                     cust=C; date=D} in dbInvoices
      and D>=yesterday;
```

where `dbItems` and `dbInvoices` are the appropriate names of tables in a relational database. The complexity in this query arises from the mapping between the relational view of the database and the 'object view' we desire. Notice how the foreign key `invNo` is crucial in connecting the two tables but is not itself present in the final result.

Queries can be embedded in speech actions. This allows them to be processed either as regular Star expressions, to be translated into SQL and processed by a normal SQL database, or to forward the entire speech action to another agent for processing.

However, mere queries do not address all issues involved in integrating an invoice processing application with a database. A typical application needs continual access to its evolving state. Querying the database periodically has severe performance and correctness issues.

What is also needed is a way of mapping updates as they occur in the database to updates in our `invoice` structure. We can achieve this may employing a transformer actor:

```
invoiceTransformer is transformer{
  invoice{
    customer = C; invoiceDate=D;
    items = all { lineItem{
      SKU = SKU; quantity=Q; price = P}
        where dbItem{ SKU=SKU; quant=Q; price=P;
                      invNo=invId} in dbItems }
  } in invoices if
      dbInvoice{ invNo=invId; cust=C;
                 date=D; } in dbInvoices and
    D>=yesterday
};
```

(We have elided some minor details here for the sake of brevity.)

This `invoiceTransformer` actor will accept updates from a source in the form of insertions, deletions and updates to tuples in the `dbInvoices` and `dbItems` relations and transform them into corresponding updates to the `invoices` relation.

Notice that the query that represents the mapping remains fundamentally the same as for the normal query. But the processing that is implied is quite different: in the case of the query the Star query is translated into an SQL query for processing by the database; in the case of the transformer updates in the underlying tables are mapped to updates in the Star data structure.

In effect, the query represents an Object-Relation mapping and the transformer represents a Relation-Object mapping. The former is similar in spirit to Spring's Hibernate; the latter does not have a widespread counterpart.

Both the query and the transformer are realized as extensions to Star using the macro language. Translating a query may involve a fairly radical transformation. For example, this query condition:

```
dbInvoice{invNo=invId;cust=C;date=D} in dbInvoices
```

is evaluated using satisfaction semantics (See Section 4.10.) Its compilation to regular evaluation involves a semantic shift.

The transformer involves another shift in semantics. In addition to the satisfaction semantics, the order of evaluation is also changed: from a top-down query evaluation to a bottom-up update-oriented semantics. We employ an approach similar to RETE [10] to achieve this transformation.

The result of using such transformations can be spectacular: This particular transformer replaces entire technologies and makes the task of interacting with databases significantly easier.

### 6.2 Semiconductor-fab scheduling

One of the first applications of the emerging Starview Enterprise Platform was *ALPS* ("Advanced Logistics and Planning System"), a system for scheduling semiconductor fabs.

Semiconductor fabrication is a complex process, with modern microprocessors going through (on the order of) 1000 production steps going on over several weeks. As modern fabs are starting to operate at 20nm resolution, the equipment is fickle and breaks down or fails to operate at specification frequently. Moreover, many fabs are moving to highly diversified product mixes. Consequently, long-term planning of the fab operation is impossible, and scheduling systems need to strike a delicate balance between the planning required to make good decisions and simultaneously not relying on the fab actually executing the plans generated.

The problem is further exacerbated by problems quite specific to semiconductors: For example, some production routes—the sequences of production steps involved in making a product—involve *queue-time zones*: A queue-time zone is a sub-sequence that, once entered, has be completed within a certain time. If a wafer does not complete a queue-time zone on time, chemical processes cause it to "go bad."

The latest iteration of the ALPS system "ALPS 4" makes use of almost all of Star's advanced facilities:

- ALPS uses Star's advanced type system to provide high-level compositional models of conceptual entities such as tools, wafers, and routes.
- Parametric polymorphism is used extensively to abstract over type definitions for concrete equipment, wafers, production recipes etc., making ALPS into a modular framework.
- The framework is *purely functional*, which enables the composition and nesting of the fab (as well as simulated fabs) with schedulers. Star enables purely functional programming with its support for higher-order programming, and through providing high-performance functional data structures for maps and sequences.
- Advanced concepts from functional programming enable the modular composition of scheduling strategies: ALPS uses monads [37] to manage the "hope" that the fab will follow its scheduling instructions, and arrows [16] to allow modular composition of scheduling strategies. Star provides general support for both through type contracts (see Section 4.6) and computation expressions (see Section 4.11).

In particular, ALPS solves the queue-time problem through the combination of its purely functional operation and the composable scheduler framework: Whenever ALPS encounters a wafer about to enter a queue-time zone, it starts a recursive speculative simulation of the system, and monitors the wafer to see if it would successfully completes the queue-time zone. Depending on the result, it lets the wafer enter or delays it.

Full speculative simulation is not possible with traditional scheduling systems used in the industry: A typical system operates on a live, multi-gigabyte database containing the fab state. Thus speculative simulation would entail copying the database state or snapshotting over longer sequences of transactions, each of which would be prohibitely expensive.

### 6.3 Transition from Java

Star is radically different from Java: It has none of the classic object-oriented program elements such as classes and objects. Moreover, Star programs typically rely on functional programming and speech actions instead of method invocations and mutable variables. Despite the differences, most Java programmers that have used Star have found the transition surprisingly easy. We do not have enough data to present an empirical analysis. However, here are some anecdotes from Star users reporting on the transition:

- As the basic organizational unit of a Java program is the class, Java is all about nouns. However, business problems are mostly about verbs.
- While Star does not have classes and objects, actors arguably solve the same problems that (at least the original notion) of objects solves: They receive messages and manage internal state in response to those messages. Speech actions are more general flexible than mere message invocation. Moreover, actors avoid the problems commonly associated with inheritance.

## 7. Comparison with other JVM languages

Today, many production-level language implementations are available for the Java platform, including a number of functional languages and languages supporting functional features. At the time development on Star started, the language landscape on the JVM was considerably more scarce, and many languages in common usage today were not yet production-ready. While originally pragmatics dictated developing a new language, Star still occupies a unique niche. In particular, most languages on the Java platform specifically target the JVM and thus retain a strong "JVM smell" even at the surface. Star, on the other hand, retains an independent feel. This section compares Star with other popular languages on the JVM that have functional elements.

***Clojure*** Clojure [14] is also a functional language, with superior support for persistent data structures, syntactic extensibility through macros, and concurrent programming with transactional memory. Support for macros in Star is similarly powerful. In contrast, Star features an "algebraic" syntax without sacrificing the generality of Lisp-style macros. Star takes a different approach to concurrency and parallelism through the asynchronous-task monad. The biggest difference is Star's sophisticated type system.

***Scala*** Scala [25] is another functional language for the JVM, with a powerful type system integrating subtyping, parametric polymorphism, existentials and a limited amount of type inference. Scala integrates object-oriented and functional features, whereas Star focusses on functional programming. This lets Star avoid some of Scala's complexities in the type system, and also makes for more uniform support for type inference. For enabling embedded DSLs, Scala relies on a variety of ingeniously conspiring but separate mechanisms: some implicit syntactic elements, block syntax, lazy fields, implicits, to name some. In contrast, Star features uniform syntactic extensibility through its macro system.

***Kotlin*** Kotlin [5] is a JVM language developed by JetBrain, the maker of the Java IntelliJ IDE. Kotlin features some functional constructs, notably first-class functions and pattern matching. However, its main design motivations are to improve upon Java, not to create a radically different language. Thus, it is still primarily an imperative object-oriented language, not a functional one. Kotlin shares Star's concern for null safety. Also, Kotlin's string templates are quite similar to Star's string interpolation.

***Groovy*** Groovy [20] started out as a dynamically-typed, more agile alternative to the JVM. As such, it features a more compact notation than Java for many programs, and also first-class functions. Groovy does offer optional static typing as well. Thus, like Kotlin, it is primarily an imperative object-oriented language. Its dynamic nature further distinguishes it from Star. Like Kotlin, Groovy shares Star's concerns about null, albeit through a null-checking operator, and offers string interpolation.

## 8. Future development

Star continues to evolve. This section sketches some of the plans for nearer-term improvements to the language:

***Macros defined by code*** Star is already homoiconic; it has a built-in representation for Star syntactic forms, and allows a program to quote an expression to obtain this representation.. We are working towards allowing macro definitions written as "regular" Star code that operate on this representation.

***High-level distributed programming*** We are developing an infrastructure for building high-level distributed applications, combining the basic model of Erlang [2] with CML-style combinators. In contrast to Erlang, we are assembling the high-level communication constructs with a combinator library to give the programmer more control over the communication behavior of message channels. We are also developing a library for distributed collections.

`invokedynamic` In Java 8, it may be feasible to use *invokedynamic* [31] and the `LambdaMetafactory` class to create closures. Significant technical challenges remain, however.

***Modules and Larger Programs*** The program-structuring tools available Star are already quite powerful. However, it currently provides only a flat namespace of *packages* to organize larger programs. A future extension to Star may include ML-style modules [32] coupled with a more explicit Agent-oriented architecture for structuring the dynamic aspects of large applications. It is anticipated that both ML-style modules and agents can be layered on top of the existing language strictly as a DSL implemented using Star's existing features of macros and contracts.

## 9. Conclusion

Star is a coherent, general-purpose programming language that combines elements from a wide variety of existing languages as well as adding innovative elements of its own. Star inherits functional programming in general, a Haskell-style type system, an F#-style monad for parallel computations, and Concurrent ML for orchestrating concurrent and parallel applications. Innovation highlights are Star's approach to syntax and macros, its speech actions for implementing agent systems, and its combination of relations, queries and satisfaction semantics. Through its sophisticated type system, extensible syntax, and macro system, Star is uniquely suited to implemented embedded DSLs. We have used Star to implement significant industrial applications, which have informed the design process. Star is an almost complete departure from Java and feels quite different to developers. Nevertheless, developers coming from Java face a surprisingly shallow learning curve.

Still, the JVM platform poses significant challenges that make compilation of Star to the JVM hard, chief amongst them the lack of proper tail calls, and size restrictions in the JVM code. The Star implementation would benefit significantly if these restrictions were lifted or at least alleviated.

***Acknowledgements*** We thank David Frese for implementing the task monad and the CML substrate. Andreas Bernauer provided valuable feedback on a draft of this paper. Also, the comments from the PPPJ reviewers were very helpful in producing the final version.

## References


[1] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*. MIT Press, 1987.

[2] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, NY, 1993. ISBN 0-13-285792-8.

[3] John L. Austin. *How to do things with words*. Oxford : Clarendon, 1962.

[4] Annette Bieniusa and Peter Thiemann. How to CPS transform a monad. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 266–280, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7.

[5] Andrey Breslav. Language of the month: Kotlin. Dr. Dobb's, `http://www.drdobbs.com/jvm/language-of-the-month-kotlin/232600836`, January 2012.

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, Grenoble, France, November 2002.

[7] William Clinger and Jonathan Rees. Macros that work. In *Proceedings 1991 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 155–162, Orlando, FL, January 1991. ACM Press.

[8] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck. *DRAFT specification of the KQML Agent Communication Language*. The DARPA knowledge sharing initiative External Interfaces Working Group, 1993.

[9] FIPA. The foundation of intelligent physical agents. `http://www.fipa.org/`.

[10] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[11] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg, FL, USA, January 1996. ACM Press. ISBN 0-89791-769-3.

[12] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[13] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Compututer Science*, 410(2-3):202–220, 2009.

[14] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic Languages*. Paphos, Cyprus, 2008.

[15] Tony Hoare. Null references: The billion dollar mistake. `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`, August 2009. Talk at QCon 2009, San Francisco.

[16] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[17] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, June 1994.

[18] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *Proceedings 4th European Symposium on Programming '92*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306, Rennes, France, February 1992. Springer-Verlag.

[19] Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Proceedings 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, Berlin, Germany, March 2000. Springer-Verlag. ISBN 3-540-67262-1.

[20] Dierk König, Guillaume Laforge, Paul King, Cédric Champeau, Hamlet D'Arcy, Erik Pragt, and Jon Skeet. *Groovy in Action*. Manning, 2nd edition, 2013.

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE, 7 Edition*. Always learning. Prentice Hall PTR, 2013. ISBN 9780133260441.

[22] David Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. John Wiley & Sons, Hoboken, New Jersey, 2012. ISBN 978-0-470-53485-4.

[23] F.G. McCabe and K.L. Clark. April - agent process interaction language. In M. Wolldridge N. Jennings, editor, *Intelligent Agents, Lecture Notes on Artificial Intelligence, vol 890*. Springer-Verlag, 1995.

[24] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0.

[25] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2nd edition, December 2010.

[26] Vaughan R. Pratt. Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 41–51, New York, NY, USA, 1973. ACM.

[27] Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In Klaus Ostermann and Walter Binder, editors, *GPCE*, pages 122–131. ACM, 2012. ISBN 978-1-4503-1129-8.

[28] John Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel Concurrent ML. *SIGPLAN Not.*, 44(9):257–268, August 2009. ISSN 0362-1340.

[29] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[30] John Rose. Tail calls in the JVM. Blog post, July 2007. `https://blogs.oracle.com/jrose/entry/tail_calls_in_the_vm`.

[31] John R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-874-2.

[32] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 89–102, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9.

[33] J. R. Searle. *Speech acts: an essay in the philosophy of language*. Cambridge University Press, 1969.

[34] Guy L. Steele. Tail calls on the JVM. Personal communication, May 1996.

[35] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, PADL'11, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18377-5.

[36] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, Berkeley, CA, USA, 2003. USENIX Association.

[37] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, May 1995.

[38] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language without even being odd. In *1998 ACM SIGPLAN Workshop on ML*, pages 24–30, Baltimore, Maryland, September 1998.

[39] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2nd edition, June 2009. ISBN 0470519460.