Type Safe Algorithmic Skeletons

Denis Caromel, Ludovic Henrio, and Mario Leyton
INRIA Sophia-Antipolis, Université de Nice Sophia-Antipolis, CNRS - I3S
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France
First.Last@sophia.inria.fr

Abstract

This paper addresses the issue of type safe algorithmic skeletons. From a theoretical perspective we contribute by: formally specifying a type system for algorithmic skeletons, and proving that the type system guarantees type safety.

From an implementation point of view, we show how it is possible to enforce the type system on an Java based algorithmic skeleton library. The enforcement takes place at the composition of the skeleton program, by typing each skeleton with respect to its construction parameters: sequential functions, and other skeletons.

As a result, hierarchical skeleton nesting can be performed safely, since type errors can be detected by the skeleton type system.

Keywords: Type systems, algorithmic skeletons.

1 Introduction

From a general point of view, typing ensures safety properties of programming languages. For instance, in object-oriented languages, typing ensures that each field or method access will reach an existing field or method: "message not understood" errors will not occur. Nevertheless, type systems are limited, that is why in most object-oriented languages, type-casts have been introduced. In exceptional cases, type-casts allow programmers to precise the type of a known object, but this step is error-prone. Therefore, it is important to reduce the number of necessary type-casts to increase the type safety of a program; this is one of the purposes of Java generics or C++ templates, for example.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [11]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the non-functional aspects regarding parallelization and distribution are implicitly defined by the composed parallel structure. Once the structure has been defined, programmers complete the program by providing the application's sequential blocks, called *muscle* functions.

Muscle functions correspond to the sequential functions of the program. We classify muscle functions using the following categories: execution (denoted f_e), evaluation of conditions (f_b), division of data (f_d), and conquest of results (f_c). Each muscle function is a black box unit for the skeleton language/framework, and while each may be error free on its own, an inadequate combination with a skeleton pattern can yield runtime typing errors. This happens because, during the evaluation of the skeleton program, the result of one muscle function is passed to "the next" muscle function as parameter. Where "the next" is determined at runtime by the specific skeleton assembly. An incompatible type between the result of a muscle function and the parameter of the next one yields runtime errors, which are difficult to detect and handle on distributed environments.

Untyped skeleton programming forces the programmer to rely on type-casts in the programming of muscle functions. Indeed, if the higher-level language (skeletons) is not able to transmit types between muscle functions, then the poorest assumption will be taken for typing muscle codes (e.g., in an untyped skeleton implementation in Java, muscle functions accept only `Object` as type for parameter/result). In that case, every object received by a muscle function has to be casted into the right type, which is highly error-prone.

On the other hand, typed skeletons relieve the programmer from having to type-cast every muscle function argument: basically, the type system will check that the input type of a skeleton is equal to the declared output type of the preceding skeleton. Type-casts remain necessary only when it would be required by the underlying language. *To summarize, the type safety ensured by the underlying language is transmitted by the skeletons: type safety is raised to the skeleton level.*

Let us consider the example shown in Figure 1, which exposes the dangers of typeless skeleton pro-

gramming. In the example two functions $\{f_1, f_2\}$ are executed sequentially the other using a *pipe* skeleton: $pipe(seq(f_1), seq(f_2))$. During the evaluation of the skeleton program, the unknown return type of `Execute1.exec` will be passed as parameter to `Execute2.exec` which is expecting a type `B` parameter. While `Execute1.exec` (f_1) and `Execute2.exec` (f_2) may be correct on their own, there is no guarantee that piping them together will not yield an error.

As shown in the example, type safe skeleton programming does not only require a method for expressing the types of muscle functions (parameters/return values), but also an underlying type-system expressing how the typed muscle functions interact with the skeleton patterns.

Indeed, one of the original aspects of the proposed type system for algorithmic skeletons is that it transmits types between muscle functions. As a consequence, the type preservation property has a greater impact than in usual type systems: it also ensures that muscle functions will receive and produce correct types relatively to the assembly.

Our contribution consists in detecting type errors in skeleton programs, thus improving the safeness of algorithmic skeleton programming. From the theoretical side, we contribute by providing type semantics rules for each of the skeleton patterns in the language. Then, we prove that these typing rules indeed satisfy type safety properties. On the practical side, we show how the proposed type system can be implemented in a Java [16] skeleton library, by taking advantage of Java Generics [7]. As such, we show that no further type-casts are imposed by the skeleton language inside muscle functions, and more importantly, that all the typing validations of skeleton compositions are performed at compilation time.

Section 2 presents related works. In section 3 we formally define a type system for skeletons and prove its correctness. Then, section 4 shows an implementation of the type system in a skeleton framework based on Java.

2 Related Work

Aldinucci et al. have provided semantics that can handle both task and data parallelism [1, 2]. The semantics describe both functional and parallel behavior of the skeleton language using a labeled transition system. Therefore, in this paper we do not focus on the parallelism aspects of the reduction semantics, and instead use big-step reduction semantics. On the other hand, this paper focuses on the typing rules, which allows us to deal with type safety.

As a skeleton framework we use Calcium [10]. Calcium is implemented in Java [16], and achieves distributed computation using ProActive [9]. ProActive is a Grid middleware that, among others, supports: an active object programming model [8], and a deployment framework [5].

Calcium is mainly inspired by Lithium [3, 13] and Muskel [12] frameworks, developed at University of Pisa. In all of them, skeletons are provided to the programmer through a Java API.

The Muesli skeleton library [14, 15] provides some of its skeletons as generics using C++ templates. Nevertheless, no type system is enforced with the templates, allowing type unsafe skeleton compositions. Concerning typing, the P3L skeleton language [4] provides type verification at the data flow level, which must be explicitly handled by the programmer using intermediate variables. Compared with Calcium, the type system proposed in Calcium enforces safeness at a higher level of abstraction: the skeleton level, where the data flow is implicit. In Skil [6], typed skeletons are used as a mean to make skeletons polymorphic. Skil translates polymorphic high order functions into monomorphic first order C functions. Nevertheless, the type system described in Skil is not a formal type system, and hence does not prove type safety of skeleton composition.

3 A typed algorithmic skeleton language

This section defines a type theory for skeleton programming. In order to present such a theory, we first specify a syntax for algorithmic skeletons in a very classical way. Then section 3.2 defines a big-step reduction semantics for skeletons; this semantics, though not as rich as the one presented in [1, 2] is sufficient for proving the type properties that interest us. We then provide a simple type system [17], and prove that this type system provides the usual property of type preservation: subject-reduction. Subject-reduction ensures that skeleton compositions do not threaten typing, and that type information can be passed by skeletons to be used in the programming of muscle function codes. This property greatly improves the correctness of skeleton programs by allowing the underlying language and the algorithmic skeletons to cooperate on this aspect.

3.1 Skeleton Language Grammar

The skeleton grammar supports several skeleton patterns. The task parallel skeletons are: seq for wrapping execution functions; $farm$ for task replication; $pipe$ for staged computation; $while/for$ for iteration; and if for conditional branching. The data parallel skeletons are: map for single instruction multiple data; $fork$ which is like map but applies multiple instructions to multiple data; and $d\&c$ for divide and conquer.

$$\begin{aligned}\triangle ::= & \; seq(f_e) \mid farm(\triangle) \mid pipe(\triangle_1, \triangle_2) \mid while(f_b, \triangle) \mid \\ & if(f_b, \triangle_{true}, \triangle_{false}) \mid for(i, \triangle) \mid map(f_d, \triangle, f_c) \mid \\ & fork(f_d, \{\triangle_i\}, f_c) \mid d\&c(f_d, f_b, \triangle, f_c)\end{aligned}$$

```
Skeleton stage1= new Seq(new Execute1());
Skeleton stage2= new Seq(new Execute2());
Skeleton skeleton=new Pipe(stage1, stage2); //type error undetected during composition
```

```
class Execute1 implements Execute{
   public Object exec(Object o){
      A a = (A)o; /* Runtime cast */
      ...
      return x; /* Unknown type */
   }
}
```

```
class Execute2 implements Execute{
   public Object exec(Object o){
      B b = (B)o; /* Runtime cast */
      ...
      return y; /* Unknown type */
   }
}
```

Figure 1. Motivation Example: Unsafe skeleton programming.

Notations In the following, f_x denotes muscle functions, $\triangle$ denotes skeletons, terms are lower case identifiers and types upper case ones. Also, brackets denote lists (or sets) of elements, i.e., $\{p_i\}$ is the list consisting of the terms p_i, and $\{Q\}$ is the type of a list of elements of type Q.

3.2 Reduction Semantics

Figure 2 presents a big-step operational semantics for skeletons. It relies on the fact that semantics for muscle functions are defined externally. In other words, for any function f_x and for any term p we have a judgment of the form: $f_x(p) \Downarrow r$.

For example, the *pipe* construct corresponds to the sequential composition of two algorithmic skeletons. Thus R-PIPE states that if a skeleton $\triangle_1$ applied to a parameter p ($\triangle_1(p)$) can be reduced to s, and $\triangle_2(s)$ can be reduced to r, then $pipe(\triangle_1, \triangle_2)(p)$ will be reduced to r. In other words, the result of $\triangle_1$ is used as parameter for $\triangle_2$: $\triangle_2(\triangle_1(p)) \Downarrow r$.

The $d\&c$ construct performs recursive divide and conquer. It recursively splits its input by the divide function f_d until the condition f_b is false, processes each piece of the divided data, and merges the results by a conquer function f_c.

3.3 Type System

Figure 3 defines a type system for skeletons. It assumes that each muscle function is of the form: $f_x : P \rightarrow R$, and verifies the following classical typing rule:

$$\text{APP-F}\quad \frac{p : P \qquad f_x : P \rightarrow R}{f_x(p) : R}$$

We first define a typing rule for each of the skeleton constructs. These typing rules allow us to infer the type of an algorithmic skeleton based on the type of the skeletons and

$$\text{R-FARM}\quad \frac{\triangle(p) \Downarrow r}{farm(\triangle)(p) \Downarrow r} \qquad \text{R-PIPE}\quad \frac{\triangle_1(p) \Downarrow s \qquad \triangle_1(s) \Downarrow r}{pipe(\triangle_1, \triangle_2) \Downarrow r}$$

$$\text{R-SEQ}\quad \frac{f_e(p) \Downarrow r}{seq(f_e)(p) \Downarrow r} \qquad \text{R-IF-TRUE}\quad \frac{f_b(p) \Downarrow true \qquad \triangle_{true}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}$$

$$\text{R-IF-FALSE}\quad \frac{f_b(p) \Downarrow false \qquad \triangle_{false}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}$$

$$\text{R-WHILE-TRUE}\quad \frac{f_b(p) \Downarrow true \qquad \triangle(p) \Downarrow s \qquad while(f_b, \triangle)(s) \Downarrow r}{while(f_b, \triangle)(p) \Downarrow r}$$

$$\text{R-WHILE-FALSE}\quad \frac{f_b(p) \Downarrow false}{while(f_b, \triangle)(p) \Downarrow p} \qquad \text{R-FOR}\quad \frac{\forall i < n \quad \triangle(p_i) \Downarrow p_{i+1}}{for(n, \triangle)(p_0) \Downarrow p_n}$$

$$\text{R-MAP}\quad \frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{map(f_d, \triangle, f_c)(p) \Downarrow r}$$

$$\text{R-FORK}\quad \frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle_i(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r}$$

$$\text{R-D\&C-FALSE}\quad \frac{f_b(p) \Downarrow false \qquad \triangle(p) \Downarrow r}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}$$

$$\text{R-D\&C-TRUE}\quad \frac{\begin{array}{c} f_b(p) \Downarrow true \qquad f_d(p) \Downarrow \{p_i\} \\ \forall i \; d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r \end{array}}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}$$

Figure 2. Skeleton's Reduction Semantics

muscle functions composing it. Typing judgments for skeletons are of the form $\triangle : P \rightarrow R$. We explain below the typing of two representative rules: T-PIPE and T-D&C.

T-PIPE Consider a *pipe* formed of skeletons $\triangle_1$ and $\triangle_2$. First, the input type of *pipe* is the same as the input type of $\triangle_1$, and the output type of *pipe* is the same as the output of $\triangle_2$. Moreover, the output type of $\triangle_1$ must match the input type of $\triangle_2$. In other words, consider the typeless skeleton example shown in Figure 1, which would be equivalent to the following code:

```
Object a = ...; //previous result
Object b = Execute1.execute(a);
Object c = Execute2.execute(b);
```

Without a typing system, the *pipe* skeleton cannot ensure type safety. The *pipe* typing ensures that types transmitted by the *pipe* parameters are compatible; and the recursive nature of the typing system ensures the correct typing of skeletons containing a *pipe*, but also of skeletons nested inside the *pipe*. Here, type compatibility ensures that the type of `b` is compatible with the type of the parameter for `Execute2.execute`. Also, that `a` is compatible with the parameter of `Execute1.execute`, and the type of `c` is compatible with the following muscle instruction (belonging to another skeleton construct).

With a typing system, the *pipe* skeleton yields a code equivalent to:

```
A a = ...; //previous muscle result
B b = Execute1.execute(a);
C c = Execute2.execute(b);
```

Where the types of $\{a, b, c\}$ are: $\{a{:}A,\ b{:}B,\ c{:}C\}$.

T-D&C Consider now a $d\&c$ skeleton, that accepts an input of type P and returns an output of type R. Therefore, the choice function f_b must also accept an input of type P, and return a boolean. Secondly, the divide function f_d produces a list of elements of P. Then, each element is computed by a sub-skeleton of type $P \rightarrow R$. Finally, the conquering function f_c must accept as input a list of elements of R and return a single element of type R, which corresponds to the return type of the $d\&c$ skeleton.

To summarize, the typing rules follow the execution principles of the skeletons, attaching a type to each intermediate result and transmitting types between skeletons.

Finally, a skeleton applied to a term follows the trivial typing rule:

$$\text{APP-}\triangle\ \frac{p : P \qquad \triangle : P \rightarrow R}{\triangle(p) : R}$$

$$\text{T-FARM}\ \frac{\triangle : P \rightarrow R}{farm(\triangle) : P \rightarrow R} \qquad \text{T-PIPE}\ \frac{\triangle_1 : P \rightarrow X \qquad \triangle_2 : X \rightarrow R}{pipe(\triangle_1, \triangle_2) : P \rightarrow R}$$

$$\text{T-SEQ}\ \frac{f_e : P \rightarrow R}{seq(f_e) : P \rightarrow R}$$

$$\text{T-IF}\ \frac{f_b : P \rightarrow boolean \qquad \triangle_{true} : P \rightarrow R \qquad \triangle_{false} : P \rightarrow R}{if(f_b, \triangle_{true}, \triangle_{false}) : P \rightarrow R}$$

$$\text{T-WHILE}\ \frac{f_b : P \rightarrow boolean \qquad \triangle : P \rightarrow P}{while(f_b, \triangle) : P \rightarrow P}$$

$$\text{T-FOR}\ \frac{i : integer \qquad \triangle : P \rightarrow P}{for(i, \triangle) : P \rightarrow P}$$

$$\text{T-MAP}\ \frac{f_d : P \rightarrow \{Q\} \qquad \triangle : Q \rightarrow S \qquad f_c : \{S\} \rightarrow R}{map(f_d, \triangle, f_c) : P \rightarrow R}$$

$$\text{T-FORK}\ \frac{f_d : P \rightarrow \{Q\} \qquad \triangle_i : Q \rightarrow S \qquad f_c : \{S\} \rightarrow R}{fork(f_d, \{\triangle_i\}, f_c) : P \rightarrow R}$$

$$\text{T-D\&C}\ \frac{\begin{array}{c} f_b : P \rightarrow boolean \\ f_d : P \rightarrow \{P\} \qquad \triangle : P \rightarrow R \qquad f_c : \{R\} \rightarrow R \end{array}}{d\&c(f_d, f_b, \triangle, f_c) : P \rightarrow R}$$

Figure 3. Skeleton's Type System

3.4 Typing Property: Subject Reduction

A crucial property of typing systems is subject reduction. It asserts the type preservation by the reduction semantics; this means that a type inferred for an expression will not change (or will only become more precise) during the execution: the type of an evaluated expression is compatible with the type of this expression before evaluation. Without this property, none of the properties ensured by the type-system would be useful. For skeletons, subject-reduction can be formalized as follows.

Theorem 1 (Subject Reduction)**.** *Provided muscle functions ensure application and subject reduction, i.e.:*

$$\text{SR-F}\ \frac{f_x(p) : Q \qquad f(p) \Downarrow q}{q : Q}$$

The skeleton type system ensures subject reduction:

$$\text{SR-}\triangle\ \frac{\triangle(p) : R \qquad \triangle(p) \Downarrow r}{r : R}$$

While the property should be proven for every single skeleton construct, with conciseness in mind, we only illustrate here the proof in the representative cases of: $pipe$, for, and $d\&c$ constructs. Please refer to the appendix for the other constructs. The general structure of the proof is straightforward. For each skeleton construct we: particularize subject-reduction; decompose the inference that can lead to the correct typing of the skeleton; and verify that, for each possible reduction rule for the skeleton, the type is preserved by the reduction. The proof also involves some double recursions in the most complex cases.

To prove this property, an alternative approach can be to design a type-system closer to the one of λ-calculus (but with a fixed point operator). Nevertheless, we have chosen the operational semantics approach because of its simplicity, and direct meaning in terms of typed skeletons.

Pipe Preservation

Subject-reduction for $pipe$ skeletons means:

$$\text{SR-PIPE}\ \frac{pipe(\triangle_1, \triangle_2)(p) : R \qquad pipe(\triangle_1, \triangle_2)(p) \Downarrow r}{r : R}$$

Proof. Let us decompose the inferences asserting that $pipe(\triangle_1, \triangle_2)(p) : R$ and $pipe(\triangle_1, \triangle_2)(p) \Downarrow r$, necessarily:

$$\frac{p : P \qquad \dfrac{\triangle_1 : P \to X \qquad \triangle_2 : X \to R}{pipe(\triangle_1, \triangle_2) : P \to R}\ \text{T-PIPE}}{pipe(\triangle_1, \triangle_2)(p) : R}\ \text{APP-}\triangle$$

$$\frac{\triangle_1(p) \Downarrow x \qquad \triangle_2(x) \Downarrow r}{pipe(\triangle_1, \triangle_2)(p) \Downarrow r}\ \text{R-PIPE}$$

Finally, we prove that r has the type R as follows:

$$\dfrac{\dfrac{\dfrac{p : P \quad \triangle_1 : P \to X}{\triangle_1(p) : X} \qquad \triangle_1(p) \Downarrow x}{x : X} \qquad \triangle_2 : X \to R}{\triangle_2(x) : R} \qquad \triangle_2(s) \Downarrow r}{r : R}$$

(APP-$\triangle$, SR-$\triangle$, APP-$\triangle$, SR-$\triangle$)

To summarize, we proved the subject-reduction property for (T-PIPE) combined with (APP-$\triangle$), which is the only way to obtain a correctly typed and reducible expression involving a pipe construct. □

For Preservation

In the case of the for skeleton, we must prove:

$$\text{SR-FOR}\ \frac{for(n, \triangle)(p) : P \qquad for(n, \triangle)(p) \Downarrow r}{r : P}$$

Proof. We decompose $for(n, \triangle)(p) : P$ and $for(n, \triangle)(p) \Downarrow r$, noting $p_0 = p$, $p_n = r$ we have:

$$\frac{p_0 : P \qquad \dfrac{n : integer \quad \triangle : P \to P}{for(n, \triangle) : P \to P}\ \text{T-FOR}}{for(n, \triangle)(p_0) : P}\ \text{APP-}\triangle$$

$$\frac{\forall i < n \quad \triangle(p_i) \Downarrow p_{i+1}}{for(n, \triangle)(p_0) \Downarrow p_n}\ \text{R-FOR}$$

We prove that $\forall i \leq n, p_i : P$ using induction on i. The base case is true $p_0 : P$, the inductive hypothesis is that $p_i : P$, and we must prove that $p_{i+1} : P$. Applying the recurrence hypothesis SR-$\triangle$, and APP-$\triangle$ we have:

$$\text{APP-}\triangle\ \text{SR-}\triangle\ \frac{\dfrac{p_i : P \qquad \triangle : P \to P}{\triangle(p_i) : P} \qquad \triangle(p_i) \Downarrow p_{i+1}}{p_{i+1} : P}$$

Therefore, $p_n : P$, and in the orginal notation $r : P$. □

D&C Preservation

For $d\&c$ skeletons, subject-reduction becomes:

$$\text{SR-}d\&c\ \frac{d\&c(f_d, f_b, \triangle, f_c)(p) : R \qquad d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}{r : R}$$

Proof. $d\&c(f_d, f_b, \triangle, f_c)(p) : R$ necessarily comes from

$$\frac{p : P \qquad \dfrac{\begin{array}{c} f_b : P \to boolean \qquad f_d : P \to \{P\} \\ \triangle : P \to R \qquad f_c : \{R\} \to R \end{array}}{d\&c(f_d, f_b, \triangle, f_c) : P \to R}\ \text{T-D\&C}}{d\&c(f_d, f_b, \triangle, f_c)(p) : R}\ \text{APP-}\triangle$$

In addition to the skeleton based recurrence, we need to use another recurrence on p for which the base case is $f_b(p) \Downarrow false$ and the inductive case is $f_b(p) \Downarrow true$. This recursion is finite because the division of the problem must always terminate (one can formalize the recurrence based on a $size$ function such that $f_d(p) \Downarrow \{p_i\} \Rightarrow size(p_i) < size(p)$ and $size(p) \leq 0 \Rightarrow f_b(p) \Downarrow false$).

CASE 1: if $f_b(p) \Downarrow false$

$$\frac{f_b(p) \Downarrow false \qquad \triangle(p) \Downarrow r}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}\ \text{R-}D\&C\text{-FALSE}$$

by hypothesis, $\triangle : P \to R$ and thus:

$$\text{APP-}\triangle\ \frac{\triangle : P \to R \qquad p : P}{\triangle(p) : R}$$

By the recurrence hypothesis ($\triangle$ is sub-skeleton of $d\&c(f_d, f_b, \triangle, f_c)$), $\triangle$ verifies subject reduction:

$$\frac{\triangle(p) : R \qquad \triangle(p) \Downarrow r}{r : R}\ \text{SR-}\triangle$$

which ensures that $r : R$ and the subject reduction for $d\&c$.

CASE 2: if $f_b(p) \Downarrow true$

$$\frac{\forall i \quad \dfrac{f_b(p) \Downarrow true \qquad f_d(p) \Downarrow \{p_i\}}{d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}\ \text{R-D\&C-TRUE}$$

First, each p_i is of type P:

$$\text{APP-F}\ \frac{f_d : P \to \{P\} \qquad p : P}{f_d(p) : \{P\}} \qquad \text{SR-F}\ \frac{f_d(p) : \{P\} \qquad f_d(p) \Downarrow \{p_i\}}{\{p_i\} : \{P\}}$$

and thus, using the "sub"recurrence hypothesis, that is subject reduction on p_i:

$$\frac{d\&c(f_d, f_b, \triangle, f_c)(p_i) : R \qquad d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i}{r_i : R}\ \text{SR-D\&C}$$

Therefore, $\forall i$, $r_i : R$, and then $\{r_i\} : \{R\}$. Finally $r : R$, because (by subject reduction on f_c):

$$\text{APP-F}\ \frac{f_c : \{R\} \to R \qquad \{r_i\} : \{R\}}{f_c(\{r_i\}) : R} \qquad \text{SR-F}\ \frac{f_c(\{r_i\}) : R \qquad f_c(\{r_i\}) \Downarrow r}{r : R}$$

Note that the typing rule for d&c also ensures that $f_b(p) : boolean$ (and thus $f_b(p)$ is necessarily true or false). □

3.5 Sub-typing

This section briefly discusses how sub-typing rules can be safely added to the type-system without major changes. Classically, if the underlying language supports sub-typing, see for example [17], we allow the skeleton typing to reuse the sub-typing relation of the language. Suppose a sub-typing relation ($\trianglelefteq$) is defined on the underlying language; then sub-typing can be raised to the skeleton language level as specified by the rule:

$$\text{T-SUB}\ \frac{\triangle : P \to R \qquad P' \trianglelefteq P \qquad R \trianglelefteq R'}{\triangle : P' \to R'}$$

The subject-reduction and most classical sub-typing properties can be proved like in other languages.

4 Type safe skeletons in Java

In this section, we illustrate the type-system designed above in the context of a skeleton library implemented over the Java programming language. We show how the type-system of the Java language can be used at the skeleton level to check the type-safety of the skeleton composition. We have chosen Java for an implementation because Java provides a mechanism to communicate the type of an object to the compiler: *generics*. More precisely, we use Java generics to specify our skeleton API: constraints on type compatibility expressed as typing rules in Figure 3 are translated into the fact that, in the skeleton API, several parameters have the same (generic) type.

Typing defined in Figure 3 is then ensured by the type system for Java and generics. Using generics, we do not need to implement an independent type system for algorithmic skeletons. Additionally, generics provide an elegant way to blend the type system of the skeleton language with the type system of the underlying language. In other words, because skeletons interact with muscle functions, the proposed skeleton type system also interacts with the Java language type system.

In the Calcium skeleton framework, skeletons are represented by a corresponding class, and muscle functions are identified through interfaces. A muscle function must implement one of the following interfaces: `Execute`, `Condition`, `Divide`, or `Conquer`. Instantiation of a skeleton requires as parameters the corresponding muscle functions, and/or other already instantiated skeletons.

As discussed in section 3.3, the idea behind the type semantics is that, if it is possible to guarantee that the skeleton parameters have compatible types, then the skeleton instance will be correctly typed.

Therefore, since the skeleton program is defined during the construction of the skeleton objects, the proper place for performing type validation corresponds to the constructor methods of the skeleton classes.

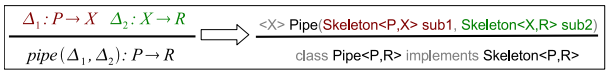

Figure 4. From theory to practice: *T-PIPE* rule.

Figure 4 shows the analogy between the type semantics and the Java implementation with generics for the *T-PIPE* rule. The premises of the typing rules are enforced in the signature of the skeleton constructor, and the conclusion of the typing is reflected on the signature of the skeleton class.

A skeleton class is now identified as a skeleton that receives a generic parameter of type `<P>` and returns a parameter of type `<R>`, where `<P>` and `<R>` are specified by the programmer. Additionally, all parameters must be coherently typed among themselves, and with the skeleton. This type coherence will be specific for each skeleton, following the rules of the proposed typing system 3.3. The typing rules are enforced in Calcium as shown in Figure 5.

As a result, the unsafe skeleton composition shown in Figure 1 is transformed into the type safe skeleton program shown in Figure 6. The constructors of `Pipe` and `Seq` enforce that the return type of `Execute1.exec` must be the same type as the parameter of `Execute2.exec`: `B`. If this is the case, then the `Pipe` skeleton will be of type `<A,C>`, where `A` is the parameter type of `Execute1.exec` and `C` is the return type of `Execute2.exec`.

```
Skeleton<A,B> stage1= new Seq<A,B>(new Execute1());
Skeleton<B,C> stage2= new Seq<B,C>(new Execute2());
Skeleton<A,C> skeleton = new Pipe<A,C>(stage1, stage2); //type safe composition
-------------------------------------------------------------------------------------
class Execute1 implements Execute<A,B>{    |   class Execute2 implements Execute<B,C>{
   public B exec(A param){                 |      public C exec(B param){
      ...      /* No cast required */      |        ...      /* No cast required */
      return x; /* instanceof B */         |        return y; /* instanceof C */
   }                                       |      }
}                                          |   }
```

Figure 6. Example of a type safe skeleton program.

```
interface Execute<P,R> extends Muscle<P,R>
  { public R exec(P param); }
interface Condition<P> extends Muscle<P,Boolean>
  { public boolean evalCondition(P param); }
interface Divide<P,X> extends Muscle<P,X[]>
  { public X[] divide(P param); }
interface Conquer<Y,R> extends Muscle<Y[],R>
  { public R conquer(Y[] param); }

class Farm<P,R> implements Skeleton<P,R>
  { public Farm(Skeleton<P,R> child); }
class Pipe<P,R> implements Skeleton<P,R>
  { <X> Pipe(Skeleton<P,X> s1, Skeleton<X,R> s2); }
class If<P,R> implements Skel<P,R>
  { public If(Condition<P> cond, Skeleton<P,R> ifsub,
      Skeleton<P,R> elsesub); }
class Seq<P,R> implements Skeleton<P,R>
  { public Seq(Execute<P,R> secCode); }
class While<P> implements Skeleton,P>
  { public While(Condition<P> cond, Skeleton<P,P> child)
      ; }
class For<P> implements Skeleton<P,P>
  { public For(int times, Skeleton,P> sub); }
class Map<P,R> implements Skeleton<P,R>
  { public <X,Y> Map(Divide<P,X> div, Skeleton<X,Y> sub,
      Conquer<Y,R> conq); }
class Fork<P,R> implements Skeleton<P,R>
  { public <X,Y> Fork(Divide<P,X> div, Skeleton<X,Y>...
      args, Conquer<Y,R> conq); }
class DaC<P,R> implements Skeleton<P,R>
  { public DaC(Divide<P,P> div, Condition<P> cond,
      Skeleton<P,R> sub, Conquer<R,R> conq); }
```

Figure 5. Typed skeletons with Java Generics

The benefits of using a type system for skeletons with Java generics are clear: no need to implement an additional type validation mechanism; no type-cast are imposed by the skeleton language inside muscle functions; and most importantly, type safe validation when composing the skeletons.

5 Conclusions

This paper has defined a type system for algorithmic skeletons. We have tackled this problem from both a theoretical and a practical approach. On the theoretical side we have contributed by: formally specifying a type system for algorithmic skeletons, and proving that this type system guarantees that types are preserved by reduction. Type preservation guarantees that *skeletons can be used to transmit types between muscle functions*.

On the practical side, we have implemented the type system using Java and generics. The type enforcements are ensured by the Java type system, and reflect the typing rules introduced in the theoretical section. Globally, this ensures the correct composition of the skeletons. As a result, we have shown that: no further type-casts are imposed by the skeleton language inside muscle functions; and most importantly, *type errors can be detected when composing the skeleton program.*

References

[1] M. Aldinucci and M. Danelutto. An operational semantics for skeletons. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003*, volume 13 of *Advances in Parallel Computing*, pages 63–70, Dresden, Germany, 2004. Elsevier.

[2] M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 2006.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[5] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deploy-

ment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[6] G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theor. Comput. Sci.*, 196(1-2):71–107, 1998.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[8] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[9] D. Caromel, C. Delbe, A. di Costanzo, and M. Leyton. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12:69–77, 2006.

[10] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.

[11] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[12] M. Danelutto. Qos in parallel programming through application managers. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.

[13] M. Danelutto and P. Teti. Lithium: A structured parallel programming enviroment in Java. In *Proc. of ICCS: International Conference on Computational Science*, volume 2330 of *LNCS*, pages 844–853. Springer Verlag, April 2002.

[14] H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a c++ skeleton library. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 122–130, New York, NY, USA, 2002. ACM Press.

[15] H. Kuchen and J. Striegnitz. Features from functional programming for a c++ skeleton library: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):739–756, 2005.

[16] Sun Microsystems. Java. http://java.sun.com.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

Appendix

This appendix details the subject reduction proofs that were not given in section 3.4.

Seq Preservation

We must prove:

$$\text{SR-SEQ}\ \frac{seq(f_e)(p) : R \qquad seq(f_e)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $seq(f_e)(p) : R$ and $seq(f_e)(p) \Downarrow r$ we obtain:

$$\text{SEQ-T}\ \text{APP-}\triangle\ \frac{f_e : P \to R}{\dfrac{seq(f_e) : P \to R \qquad p : P}{seq(f_e)(p) : R}} \qquad \frac{f_e(p) \Downarrow r}{seq(f_e)(p) \Downarrow r}\ \text{R-SEQ}$$

Applying APP-F and SR-F we obtain the following inference:

$$\text{APP-F}\ \text{SR-F}\ \frac{\dfrac{p : P \qquad f_e : P \to R}{f_e(p) : R} \qquad f_e(p) \Downarrow r}{r : R}$$

□

Farm Preservation

We must prove:

$$\text{SR-FARM}\ \frac{farm(\triangle)(p) : R \qquad farm(\triangle)(p) \Downarrow r}{r : R}$$

which is done similarly to the case *Seq* above, except that it uses APP-$\triangle$ and SR-$\triangle$ instead of APP-F and SR-F.

If Preservation

We must prove:

$$\text{SR-IF}\ \frac{if(f_b, \triangle_{true}, \triangle_{false})(p) : R \quad if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}{r : R}$$

Proof. By decomposing $if(f_b, \triangle_{true}, \triangle_{false})(p) : R$ and $if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r$ we obtain:

$$\dfrac{p : P \quad \dfrac{f_b : P \to boolean \quad \triangle_{true} : P \to R \quad \triangle_{false} : P \to R}{if(f_b, \triangle_{true}, \triangle_{false}) : P \to R}\ \text{T-IF}}{if(f_b, \triangle_{true}, \triangle_{false})(p) : R}\ \text{APP-IF}$$

$$\dfrac{f_b(p) \Downarrow \{true|false\} \quad \triangle_{\{true|false\}}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}\ \text{R-IF}$$

Applying app-$\triangle$ and SR-$\triangle$ on the right skeleton (Δ_{true} or Δ_{false}), we obtain the following inference:

$$\dfrac{\dfrac{p : P \quad \triangle\{true|false\} : P \to R}{\triangle_{\{true|false\}}(p) : R}\ \text{APP-}\triangle \qquad \triangle_{\{true|false\}}(p) \Downarrow r}{r : R}\ \text{SR-}\triangle$$

□

While Preservation

We must prove:

$$\dfrac{\text{SR-WHILE}\quad while(f_b, \triangle)(p) : P \qquad while(f_b, \triangle)(p) \Downarrow r}{r : P}$$

Proof. By decomposing $while(f_b, \triangle)(p) : P$, and $while(f_b, \triangle)(p) \Downarrow r$ we obtain:

$$\dfrac{p : P \quad \dfrac{f_b : P \to boolean \qquad \triangle : P \to P}{while(f_b, \triangle) : P \to P}\ \text{T-WHILE}}{while(f_b, \triangle)(p) : P}\ \text{APP-}\triangle$$

$$\dfrac{f_b(p) \Downarrow false}{while(f_b, \triangle)(p) \Downarrow p}\ \text{R-WHILE-FALSE}$$

$$\dfrac{f_b(p) \Downarrow true \quad \triangle(p) \Downarrow s \quad while(f_b, \triangle)(s) \Downarrow r}{while(f_b, \triangle)(p) \Downarrow r}\ \text{R-WHILE-TRUE}$$

This is done again by a sub-recurrence on the number of times the skeleton Δ is executed. The case where $f_b(p) \Downarrow false$ is trivial because $p : P$. For the case where $f_b(p) \Downarrow true$, we must first determine that $s : P$. Using the recurrence hypothesis (SR-$\triangle$) and app-$\triangle$:

$$\text{APP-}\triangle\ \dfrac{p : P \qquad \triangle : P \to P}{\triangle(p) : P \qquad \triangle(p) \Downarrow s}\ \text{SR-}\triangle$$

Now, by applying the sub-recurrence hypothesis on s we have:

$$\begin{array}{r}\text{APP-WHILE}\\ \text{SR-WHILE}\end{array}\ \dfrac{s : P \qquad while(f_b, \triangle) : P \to P}{while(f_b, \triangle)(s) : P \qquad while(f_b, \triangle)(s) \Downarrow r}{r : P}$$

□

Map Preservation

We must prove:

$$\dfrac{\text{SR-MAP}\quad map(f_d, \triangle, f_c)(p) : R \qquad map(f_d, \triangle, f_c)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $map(f_d, \triangle, f_c)(p) : R$ and $map(f_d, \triangle, f_c)(p) \Downarrow r$ we obtain:

$$\dfrac{p : P \quad \dfrac{f_d : P \to \{Q\} \qquad \triangle : Q \to S \qquad f_c : \{S\} \to R}{map(f_d, \triangle, f_c) : P \to R}\ \text{T-MAP}}{map(f_d, \triangle, f_c)(p) : R}\ \text{APP-}\triangle$$

$$\dfrac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{map(f_d, \triangle, f_c)(p) \Downarrow r}\ \text{R-MAP}$$

Applying APP-F and SR-F:

$$\begin{array}{r}\text{APP-F}\\ \text{SR-F}\end{array}\ \dfrac{p : P \qquad f_d : P \to \{Q\}}{f_d(p) : \{Q\} \qquad f_d(p) \Downarrow \{p_i\}}{\{p_i\} : \{Q\}}$$

Therefore $p_i : Q$, and applying APP-$\triangle$ SR-$\triangle$:

$$\begin{array}{r}\text{APP-}\triangle\\ \text{SR-}\triangle\end{array}\ \dfrac{p_i : Q \qquad \triangle : Q \to S}{\triangle(p_i) : S \qquad \triangle(p_i) \Downarrow r_i}{r_i : S}$$

Thus, $\{r_i\} : \{S\}$, and by APP-F and SR-F:

$$\begin{array}{r}\text{APP-F}\\ \text{SR-F}\end{array}\ \dfrac{\{r_i\} : \{S\} \qquad f_c : \{S\} \to R}{f_c(\{r_i\}) : R \qquad f_c(\{r_i\}) \Downarrow r}{r : R}$$

□

Fork Preservation

We must prove:

$$\dfrac{\text{SR-FORK}\quad fork(f_d, \{\triangle_i\}, f_c)(p) : R \qquad fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $fork(f_d, \{\triangle_i\}, f_c) : R$ and $fork(f_d, \{\triangle_i\}, f_c) \Downarrow r$ we obtain:

$$\dfrac{p : P \quad \dfrac{f_d : P \to \{Q\} \quad \triangle_i : Q \to S \quad f_c : \{S\} \to R}{fork(f_d, \{\triangle_i\}, f_c) : P \to R}\ \text{T-FORK}}{fork(f_d, \{\triangle_i\}, f_c)(p) : R}\ \text{APP-}\triangle$$

$$\dfrac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle_i(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r}\ \text{R-FORK}$$

Applying APP-F and SR-F:

$$\begin{array}{r}\text{APP-F}\\ \text{SR-F}\end{array}\ \dfrac{p : P \qquad f_d : P \to \{Q\}}{f_d(p) : \{Q\} \qquad f_d(p) \Downarrow \{p_i\}}{\{p_i\} : \{Q\}}$$

Then $p_i : Q$, and applying APP-$\triangle$ SR-$\triangle$, we have for all i:

$$\begin{array}{r}\text{APP-}\triangle\\ \text{SR-}\triangle\end{array}\ \dfrac{p_i : Q \qquad \triangle_i : Q \to S}{\triangle_i(p_i) : S \qquad \triangle_i(p_i) \Downarrow r_i}{r_i : S}$$

Therefore, $\{r_i\} : \{S\}$, and from APP-F and SR-F:

$$\begin{array}{r}\text{APP-F}\\ \text{SR-F}\end{array}\ \dfrac{\{r_i\} : \{S\} \qquad f_c : \{S\} \to R}{f_c(\{r_i\}) : R \qquad f_c(\{r_i\}) \Downarrow r}{r : R}$$

□