

Overloading in Tiger

Valentin “Jesus” David <valentin.david@lrde.epita.fr>

LRDE seminar, May 21, 2003



Table of Contents

What we want to overload	4
C++-like	5
Return value type using	9
Extending to variables	21
Design	26
Type Checking	28
Existing algorithms	39
Renaming	42
Recreation	44

References **55**

Part I

What we want to overload

What we want to overload

- Tiger(Appel, 1998) language.
- What can be overloaded?
- Adhoc overloading.

C++-like

Parameter types make the function signature.

```
int f(int);  
int f(char *);
```

C++ implementation

Common C++ compilers add this signature in the symbol names.

g++-2.95, como:

```
f__Fi -> int f(int);  
f__FPc -> int f(char *);
```

g++-3.2, icc:

```
_Z1fi -> int f(int);  
_Z1fPc -> int f(char *);
```

Program transformation possible implementation

We have this source:

```
let
  function f(i : int) = ()
  function f(s : string) = ()
in
  f(0);
  f("toto")
end
```

Program transformation possible implementation

It is hoped to be transform to it:

```
let
  function f_int(i : int) = ()
  function f_string(s : string) = ()
in
  f_int(0);
  f_string("toto")
end
```

Return value type using

- Not so implemented because of ambiguities.
- Some language like Ada already use it.

Ada example

```
with Text_IO; use Text_IO;
procedure Overload is
  function F return Integer is
  begin
    Text_IO.Put_Line("f:_integer");
    return 0;
  end;
  function F return String is
  begin
    Text_IO.Put_Line("f:_string");
    return "toto";
  end;
  I : Integer;
begin
  I := F;
end;
```

Tiger transformation

We have this program:

```
let
  function hello() : string =
    "hello"
  function hello() : int =
    1
in
  print(hello());
  printint(hello());
end
```

Tiger transformation

It would be transformed as:

```
let
  function hello() : string =
    "hello"
  function hello_() : int =
    1
in
  print(hello());
  printint(hello_());
end
```

Ambiguities

```
let
  function f(i : int) = ()
  function f(s : string) = ()
  function g() : int = 0
  function g() : string = "test"
in
  f(g())
end
```

Ambiguities

Execution of this could be:

- `g : int`
`f : int -> void`

or

- `g : string`
`f : string -> void`

Forced resolution

Some of the ambiguities are not solvable.

- If the developer did not tell expressively what to do, he should be warned and the result code should not be produced.
- If the developer knows what to do, he has to say it by forcing the typing.

Ada

```
procedure Overload is
  procedure F (I : Integer) is begin end;
  procedure F (S : String) is begin end;
  function G return Integer is begin return 0; end;
  function G return String is begin return "toto"; end;
begin
  F(G);
end;
```

Ada

```
procedure Overload is
  procedure F (I : Integer) is begin end;
  procedure F (S : String) is begin end;
  function G return Integer is begin return 0; end;
  function G return String is begin return "toto"; end;
begin
  F(G);
end;
```

ambiguous expression (cannot resolve "F")

Ada

```
procedure Overload is
  procedure F (I : Integer) is begin end;
  procedure F (S : String) is begin end;
  function G return Integer is begin return 0; end;
  function G return String is begin return "toto"; end;
begin
  F(String'(G));
end;
```

Tiger

```
function string(string : string) : string = string
function int(int : int) : int = int
```

Tiger

```
function string(string : string) : string = string
function int(int : int) : int = int

f(string(g()))
```

Extending to variables

Variables can be viewed as functions:

- `v` : `type` when using the value
- `v` : `type -> void` on assignment

Example

```
let
  var a := 0
  var a := "toto"
in
  print(a);
  printint(a)
end
```

would print out toto0

Field name of record selection

```
let
  type t1 = { a : int }
  type t2 = { b : int }
  var a := t1{a=0}
  var a := t2{b=1}
in
  printint(a.a);
  printint(a.b)
end
```

would print out : 01

Field type of record selection

```
let
  type t1 = { a : int }
  type t2 = { a : string }
  var a := t1{a=0}
  var a := t2{a="toto"}
in
  printint(a.a);
  print(a.a)
end
```

would print out : 0toto

Part II

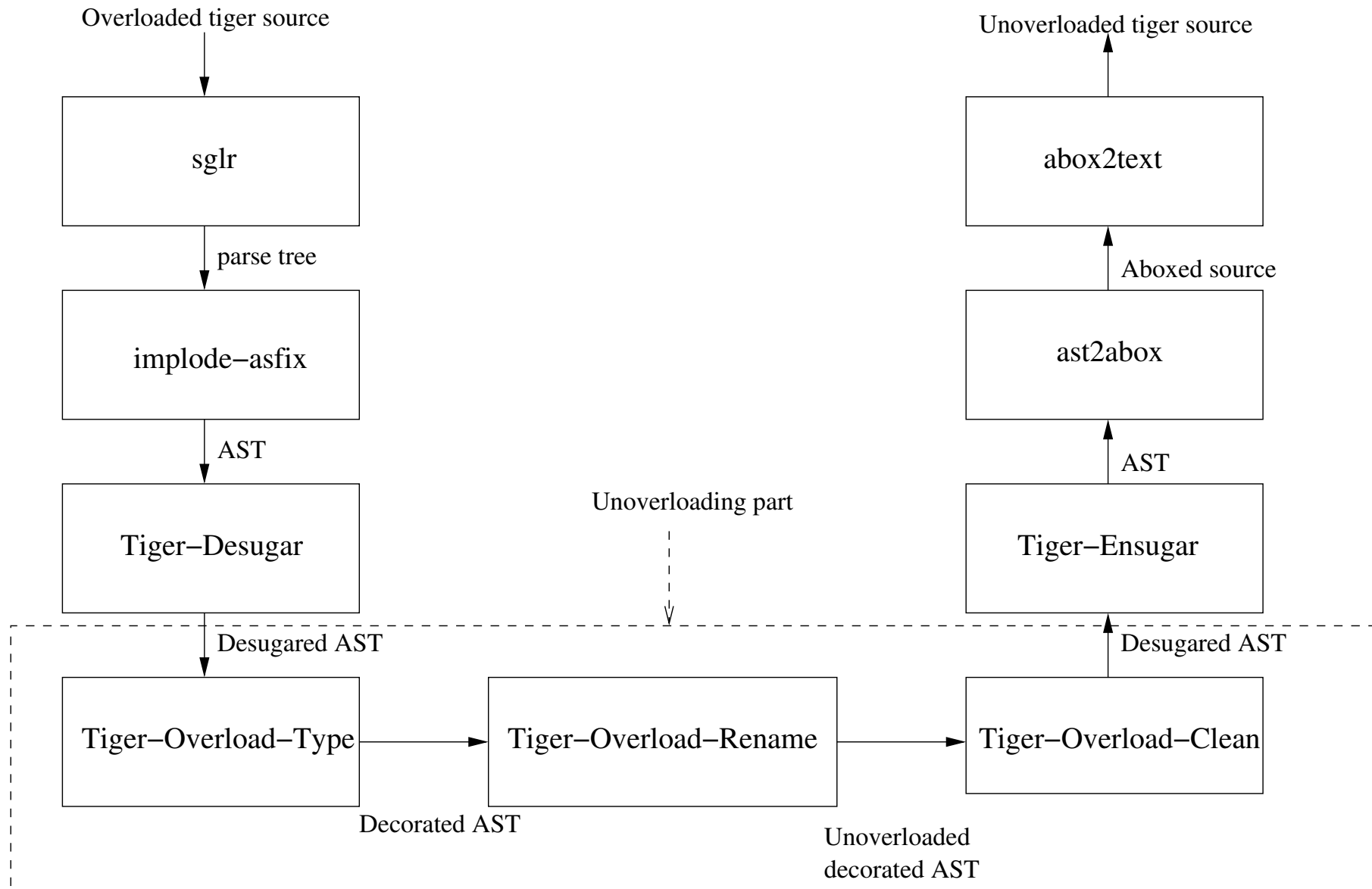
Design

Design

Using tools :

- Sdf environment ([Visser, 1997](#))
- StrategoXT ([Visser, 2001](#))
- Tiger in Stratego ([UU, 2003](#))

Design



Type Checking

- decoration of the AST with candidate type sets.
- programming with intersection of candidate type sets.

Intersections

let

```
type a = {a : int}  
function g() : int = 0  
function g() : string = "toto"  
function f(i : int) = ()  
function f(i : a) = ()
```

in

```
f(g())
```

end

Intersections

- $g()$ is typed with $G = \{int, string\}$.
- f is waiting for an argument of $F = \{int, a\}$.
- $F \cap G = \{int\}$

Field record access

Let the transformation :

Type : Typed([[lv . y]], ask) -> Typed([[lv' . y]], t)

1. Type lv
2. Get all possible fields for all the candidate types.
3. Select fields which match the name of y
4. Select fields which type is in ask
5. In possible types of lv , select records which have a previous selected field. Call the new typed left value lv' and the possible fields type t .

Function call [1/2]

Let the transformation :

Type : $\text{Typed}(\text{Call}(\text{Var}(f), e^*), \text{ask}) \rightarrow \text{Typed}(\text{Call}(\text{Var}(f), e_2^*), t)$

1. Type each argument of e^* , call it e_1^*
2. Let l , the list of types in e_1^*
3. Let $l' = l_1 \times l_2 \times \dots \times l_i$
4. Let sig_s , the declared signatures of f in σ_f
5. Let l_{sig_s} , left part of the signatures where the return value type belongs to ask

Function call [2/2]

6. Let $l'' = l' \cap lsigs$
7. For each signatures in l'' , make the necessary intersections on arguments.
8. In $sigs$ find all signatures where the parameter signatures is in l'' , deduce t

Example [1/3]

```
let
  type a = {a : int}
  function g() : int = 0
  function g() : string = "toto"
  function f(i : int, j : int) = ()
  function f(i : a, j : int) = ()
in
  f(g(), 0)
end
```

AST for the `f` call node:

```
Typed(Call(Var(f), [Call(Var(g), []), Int(0)]),
  [INT, RECORD("a", FIELD("a", INT)), STRING, UNIT])
```

Example [2/3]

- We type the arguments, the result is:

```
[Typed(Call(Var(g), []), [INT, STRING]), Typed(Int(0), [INT])]
```

- The type list is:

```
[[INT, STRING], [INT]]
```

- The result of the Cartesian product is:

```
[[INT, INT], [STRING, INT]]
```

- The available signatures of f are:

```
[[INT, INT] -> UNIT, [RECORD("a", FIELD("a", INT)), INT] -> UNIT]
```

- Only left parts of these are taken:

```
[[INT, INT], [RECORD("a", FIELD("a", INT)), INT]]
```

Example [3/3]

- The intersection gives: `[[INT, INT]]`
- The necessary intersections on arguments are made:
`[Typed(Call(Var(g), []), [INT]), Typed(Int(0), [INT])]`
- The final signatures available for `f` are selected:
`[[INT, INT] -> UNIT]`
- Return types are extracted : `[UNIT]`

The result is:

```
Typed(Call(Var(f), [Typed(Call(Var(g), []), [INT]),  
                    Typed(Int(0), [INT])]), [UNIT])
```

Characteristics of this algorithm

- Converges (quickly)
- Multi-pass
- Each pass, traverse twice each argument.
- Easy to locate the errors.

Decorated result tree

For each candidate set t^* in the decorated tree :

- If $t^* = \emptyset$, there is a type error.
- If $\text{card}(t^*) \geq 2$, there is an ambiguity

Existing algorithms

There are two one pass algorithms :

- Baker, bottom-up
- Cormack, top-down

Cormack

Let t a type. Let a function call. We check if it is possible to interpret this call when it returns a t value.

1. For each interpretation for the function (all declaration), where the return value type is t ,
 - (a) If the parameter number of this interpretation is equal to the number of argument of the call,
 - i. Check if all the arguments can be typed like the interpretation.
 - ii. If it can, it is a solution.
2. If there was a solution, the type t is possible to interpret this call.

Problems

These algorithms test all the interpretations of the code.

A problem is, that these algorithms traverse each argument for each interpretation. So, it is expensive for programs with many overloaded functions.

Another problem is, that it is hard to have a good error tracker in.

Renaming

Now, the tree has been decorated. The program has to be transformed.

Renaming

- For each function or variable declaration, if the name already exists, but for another type of function (respectively variable) rename it
 - For all these renaming, keep it in a translation table.
- For each function or variable using, look for the eventually new name in the table.

Recreation

let

```
// rational numbers and its tools
type q = { num : int, den : int }
function modulo(i1 : int, i2 : int) : int =
  i1 - ((i1 / i2) * i2)
function simplify(q : q) : q =
  let
    function pgcd(i1 : int, i2 : int) : int =
      if i2 = 0 then
        i1
      else
        if i2 > i1 then
          pgcd(i2, i1)
        else
          pgcd(i2, i1 - i2)
    var pgcd := pgcd(q.num, q.den)
  in
    q{num = q.num / pgcd, den = q.den / pgcd}
end
```

```
// addition definitions
function add(i1 : int, i2 : int) : int = i1 + i2
function add(q1 : q, q2 : q) : q = simplify(
  q{
    num = q1.num * q2.den + q1.den * q2.num,
    den = q1.den * q2.den
  })

// overloaded additions
function add(i : int, q : q) : q = add(q(i), q)
function add(q : q, i : int) : q = add(q, q(i))
function add(i : int, s : string) : q = add(q(i), q(s))
function add(q : q, s : string) : q = add(q, q(s))
function add(s : string, i : int) : q = add(q(s), q(i))
function add(s : string, q : q) : q = add(q(s), q)
function add(s1 : string, s2 : string) : q = add(q(s1), q(s2))
```

Recreation

```
// subtraction definitions
function sub(i1 : int, i2 : int) : int = i1 - i2
function sub(q1 : q, q2 : q) : q = simplify(
  q{
    num = q1.num * q2.den - q1.den * q2.num,
    den = q1.den * q2.den
  })

// overloaded subtractions
function sub(i : int, q : q) : q = sub(q(i), q)
function sub(q : q, i : int) : q = sub(q, q(i))
function sub(i : int, s : string) : q = sub(q(i), q(s))
function sub(q : q, s : string) : q = sub(q, q(s))
function sub(s : string, i : int) : q = sub(q(s), q(i))
function sub(s : string, q : q) : q = sub(q(s), q)
function sub(s1 : string, s2 : string) : q = sub(q(s1), q(s2))
```

Recreation

```
// multiplication definitions
function mul(i1 : int, i2 : int) : int = i1 * i2
function mul(q1 : q, q2 : q) : q = simplify(
  q{ num = q1.num * q2.num, den = q1.den * q2.den })

// overloaded multiplications
function mul(i : int, q : q) : q = mul(q(i), q)
function mul(q : q, i : int) : q = mul(q, q(i))
function mul(i : int, s : string) : q = mul(q(i), q(s))
function mul(q : q, s : string) : q = mul(q, q(s))
function mul(s : string, i : int) : q = mul(q(s), q(i))
function mul(s : string, q : q) : q = mul(q(s), q)
function mul(s1 : string, s2 : string) : q = mul(q(s1), q(s2))
```

Recreation

```
// division definitions
function div(i1 : int, i2 : int) : q = div(q(i1), q(i2))
function div(q1 : q, q2 : q) : q = simplify(
  q{ num = q1.num * q2.den, den = q1.den * q2.num })

// overloaded divisions
function div(i : int, q : q) : q = div(q(i), q)
function div(q : q, i : int) : q = div(q, q(i))
function div(i : int, s : string) : q = div(q(i), q(s))
function div(q : q, s : string) : q = div(q, q(s))
function div(s : string, i : int) : q = div(q(s), q(i))
function div(s : string, q : q) : q = div(q(s), q)
function div(s1 : string, s2 : string) : q = div(q(s1), q(s2))
```

Recreation

```
// conversion tools
function str2int(s : string) : int =
  if size(s) then
    str2int(substring(s, 0, size(s) - 1)) * 10
    + ord(substring(s, size(s) - 1, 1)) - ord("0")
  else
    0

function int2str_rec(i : int) : string =
  if i then
    concat(int2str_rec(i / 10), chr(modulo(i, 10) + ord("0")))
  else
    ""
```

Recreation

```
// conversion tools
function int2str(i : int) : string =
  if i then
    if i < 0 then
      concat("-", int2str_rec(-i))
    else int2str_rec(i)
  else "0"

function strnum(s : string) : string =
  if (size(s) = 0) | (substring(s, 0, 1) = "/") then ""
  else
    concat(substring(s, 0, 1), strnum(substring(s, 1, size(s)-1)))

function strden(s : string) : string =
  if size(s) = 0 then "1"
  else
    if substring(s, 0, 1) = "/" then
      substring(s, 1, size(s) - 1)
    else
      strden(substring(s, 1, size(s) - 1))
```

Recreation

```
// automatic casts
function cast(q : q) : int = q.num / q.den
function cast(i : int) : int = i
function cast(s : string) : int = str2int(s)
function cast(q : q) : string =
  concat(int2str(q.num),
    if q.den = 1 then "" else concat("/", int2str(q.den)))
function cast(i : int) : string = int2str(i)
function cast(s : string) : string = s
function cast(q : q) : q = q
function cast(i : int) : q = q { num = i, den = 1 }
function cast(s : string) : q =
  q { num = str2int(strnum(s)), den = str2int(strden(s)) }
```

Recreation

```
// force casts
function string(s : string) : string = s
function string(i : int) : string = cast(i)
function string(q : q) : string = cast(q)
function int(s : string) : int = cast(s)
function int(i : int) : int = i
function int(q : q) : int = cast(q)
function q(s : string) : q = cast(s)
function q(i : int) : q = cast(i)
function q(q : q) : q = q
```

Recreation

```
// overload of "print"  
function print(i : int) =  
    print(string(i))  
function print(q : q) =  
    print(string(q))
```

Recreation

```
in
  print(div("58", "22"));
  print("\n");
  print(div(58, q(22)));
  print("\n");
  print(div("58", q(22)));
  print("\n");
  print(div(58, "22"));
  print("\n")
end
```

References

- Appel, A. W. (1998). *Modern compiler implementation in ML*. Cambridge university press.
- UU (2003). Tiger in Stratego: An experiment in compilation by transformation. www.stratego-language.org/twiki/bin/view/Tiger.
- Visser, E. (1997). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam.
- Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.