Types for Data Races

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

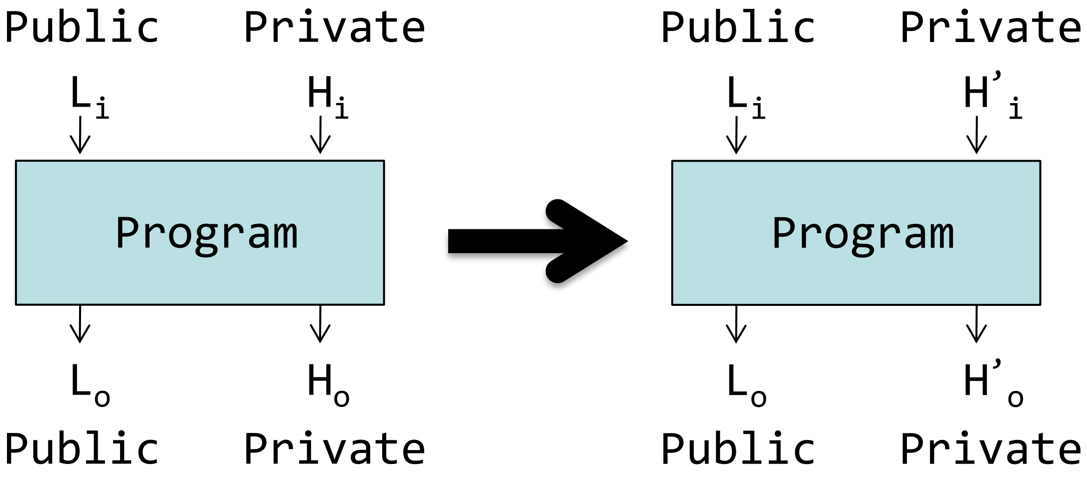
With slides by Flanagan and Freund.
Used with permission.

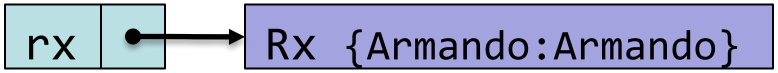
October 19, 2015

Recap

A change in a private input can not affect a public output

Data with a label L_h can not be written to a location with label L_l if $L_l <= L_h$





```
Wikipedia wp = getWP();
~~wp.write(rx);~~

  Wikipedia{
          void write(String{} txt);
  }
```

Data Races

```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```

Data Race:

Two threads access the same memory location,
one of the accesses is a write,
and there is no synchronization in between.

Strategy

How do programmers avoid races?

- Only access shared data while holding the “right” lock
  - all threads must agree on what the right lock for a piece of data is
- The decision of what the right lock is should be easy to describe
  - otherwise it’s easy to get confused

We can make this into a safety policy!

Strategy

In order to avoid races, we will design a type system to enforce the following safety property:

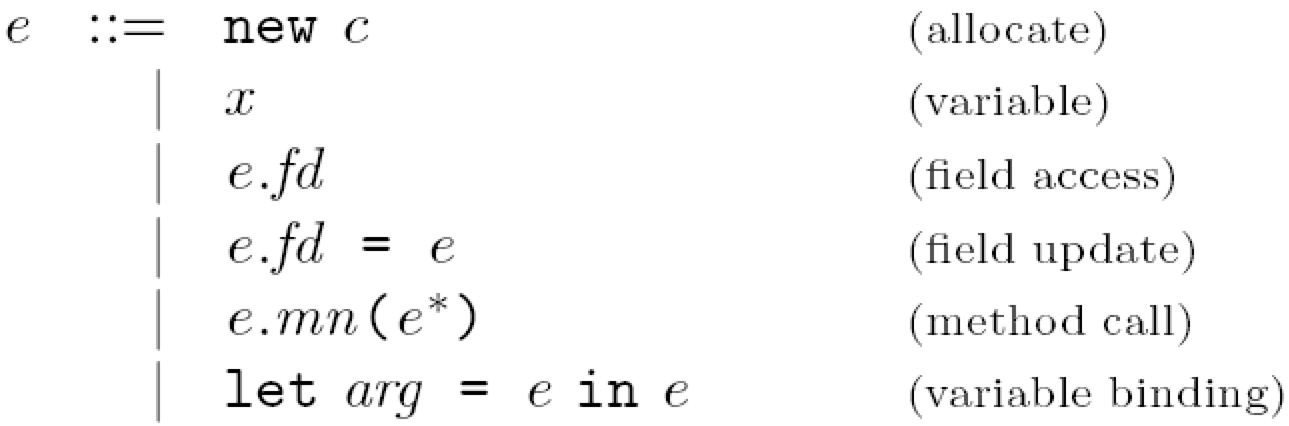
- When a memory location L is accessed by a thread, the set of locks held by the thread must be a superset of the set of locks that protect L.

Challenges:

- Define mechanisms to encode the locks that guard a memory location as part of the type
- Define a type checking algorithm that compares the required locks against a conservative approximation of the set of locks held at a given point in the program
- Define a type inference algorithm that can save you from writing lots of annotations

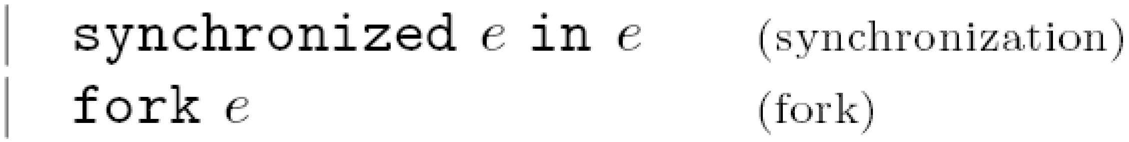
The language

Start with a simple language with classes and references



$$
\begin{array}{lcll}
e & ::= & \texttt{new}\ c & \text{(allocate)} \\
 & | & x & \text{(variable)} \\
 & | & e.fd & \text{(field access)} \\
 & | & e.fd\ \texttt{=}\ e & \text{(field update)} \\
 & | & e.mn\texttt{(}e^{*}\texttt{)} & \text{(method call)} \\
 & | & \texttt{let}\ arg\ \texttt{=}\ e\ \texttt{in}\ e & \text{(variable binding)}
\end{array}
$$

Add threads and synchronization

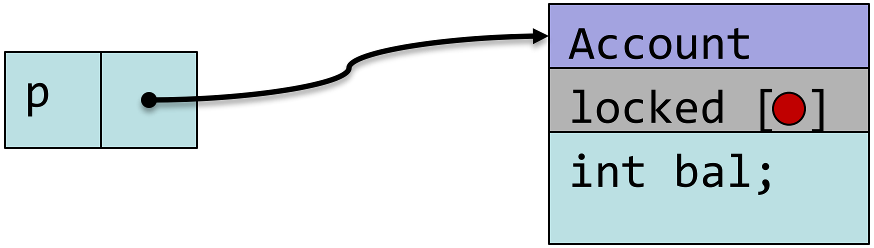


$$
\begin{array}{lll}
| & \texttt{synchronized}\ e\ \texttt{in}\ e & \text{(synchronization)} \\
| & \texttt{fork}\ e & \text{(fork)}
\end{array}
$$

Java synchronization

Every object has a lock associated with it

A synchronized block acquires and releases the lock of an object



```
...
synchronized(p){


}
...
```

We can describe sets of locks by describing sets of objects!

Stating Locking Requirements

```
class Account {
  private int bal guarded_by this = 0;

  public void deposit(int n) requires this{
    int j = bal;
    bal = j + n;
  }
}
```

Stating Locking Requirements

```
class Account {
  private int bal guarded_by this = 0;

  public void deposit(int n) requires this{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account r) requires        {
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

Stating Locking Requirements

```
class Account {
  private Guard g
  private int bal guarded_by g = 0;

  public void deposit(int n) requires g{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account r) requires g, r.g{
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

Stating Locking Requirements

```
class Account {
  private final Guard g;
  private int bal guarded_by g = 0;

  public void deposit(int n) requires g{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account r) requires g, r.g{
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

These expressions need to be final.

Stating Locking Requirements

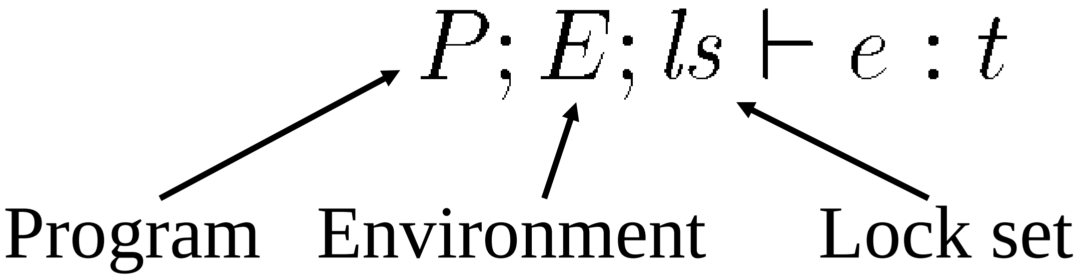
```
class Account<Ghost l> {
  private int bal guarded_by l = 0;

  public void deposit(int n) requires l{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account<l> r) requires l{
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

Type Checking

Lock Set must be included as part of the environment



$$P; E; ls \vdash e : t$$

Program Environment Lock set

Type Checking

```
class Account {
  private int bal guarded_by this = 0;

  public void deposit(int n) requires this{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account r) requires this, r{
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

```
{
   Account a = getAccnt(10220);
   Account b = getAccnt(22123);
      synchronized(a,b){
         a.transferAll(b);
      }
}
```

```
class Account {
  private final Guard g;
  private int bal guarded_by g = 0;

  public void deposit(int n) requires g{
    int j = bal;
    bal = j + n;
  }

  public void transferAll(Account r) requires g, r.g{
    int j = bal;
    int k = r.bal;
    bal = j+k;
    r.bal = 0;
  }
}
```

```
{
   Account a = getAccnt(10220);
   Account b = getAccnt(22123);
      synchronized(a.g,b.g){
         a.transferAll(b);
      }
}
```

Typing Rules

[EXP FORK]

$$\frac{P; E; \emptyset \vdash e : t}{P; E; ls \vdash \texttt{fork}\ e : \texttt{int}}$$

Typing Rules

[EXP SYNC]

$$\frac{P; E \vdash_{\text{final}} e_1 : c \qquad P; E; ls \cup \{e_1\} \vdash e_2 : t}{P; E; ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

Typing Rules

[METHOD]

$$\frac{}{P; E \vdash t\ \ mn(arg_{1\ldots n})\ \texttt{requires}\ ls\ \{\ e\ \}}$$

Typing Rules

[EXP REF]

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash ([\texttt{final}]_{\text{opt}}\ t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e'') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E; ls \vdash e' : [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd = e' : [e/\texttt{this}]t}$$

Example

```
class Node<ghost l>{
   Node<l> next guarded_by l;
   int v guarded_by l;
}

class List{
   Node<this> head

   void add(int x) requires this{
       Node<this> t = new Node<this>(x);
       t.next = head;
       head = t;
   }
}

{      List l = getList();
       synchronized(l){ l.add(5); }
}
```

Type Inference

How do we avoid adding all of these annotations?

Reducing Type Inference to SAT

```
class Ref<ghost g1,g2,...,gn> {
  int i;
  void add(Ref r)


  {
     i = i
         + r.i;
  }
}
```

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i;
  void add(Ref r)

  {
     i = i
         + r.i;
  }
}
```

- Add ghost parameters <ghost g> to each class declaration

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref r)

  {
    i = i
        + r.i;
  }
}
```

- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by $\alpha_1$;
  void add(Ref<$\alpha_2$> r)

  {
    i = i
        + r.i;
  }
}
```

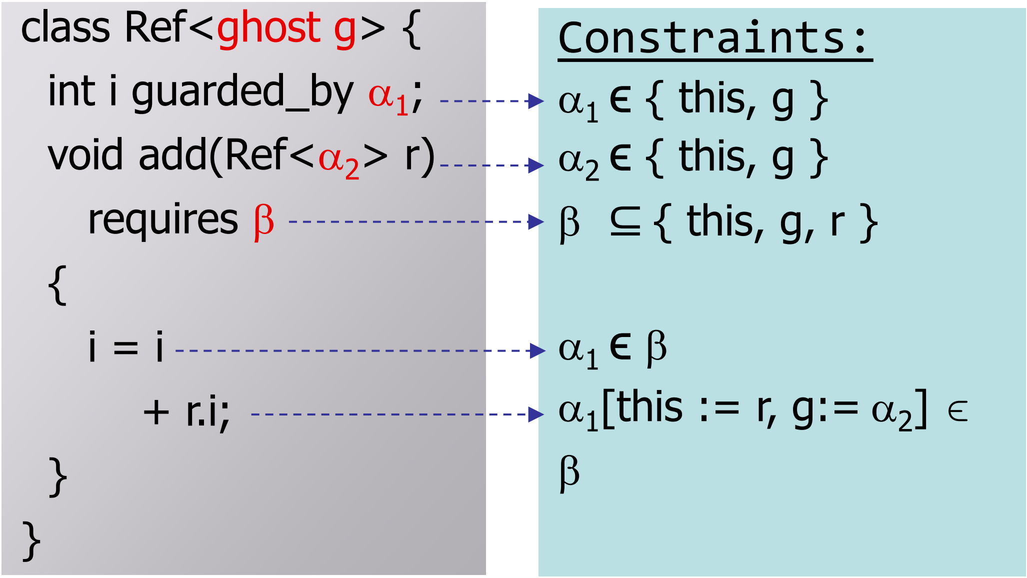
- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock
- Add <α_2> to each class reference

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by $\alpha_1$;
  void add(Ref<$\alpha_2$> r)
    requires $\beta$
  {
    i = i
        + r.i;
  }
}
```

- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock
- Add <α_2> to each class reference
- Add requires β_i to each method
  - type inference resolves β_i to some set of locks

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by α_1;
  void add(Ref<α_2> r)
     requires β
  {
     i = i
         + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

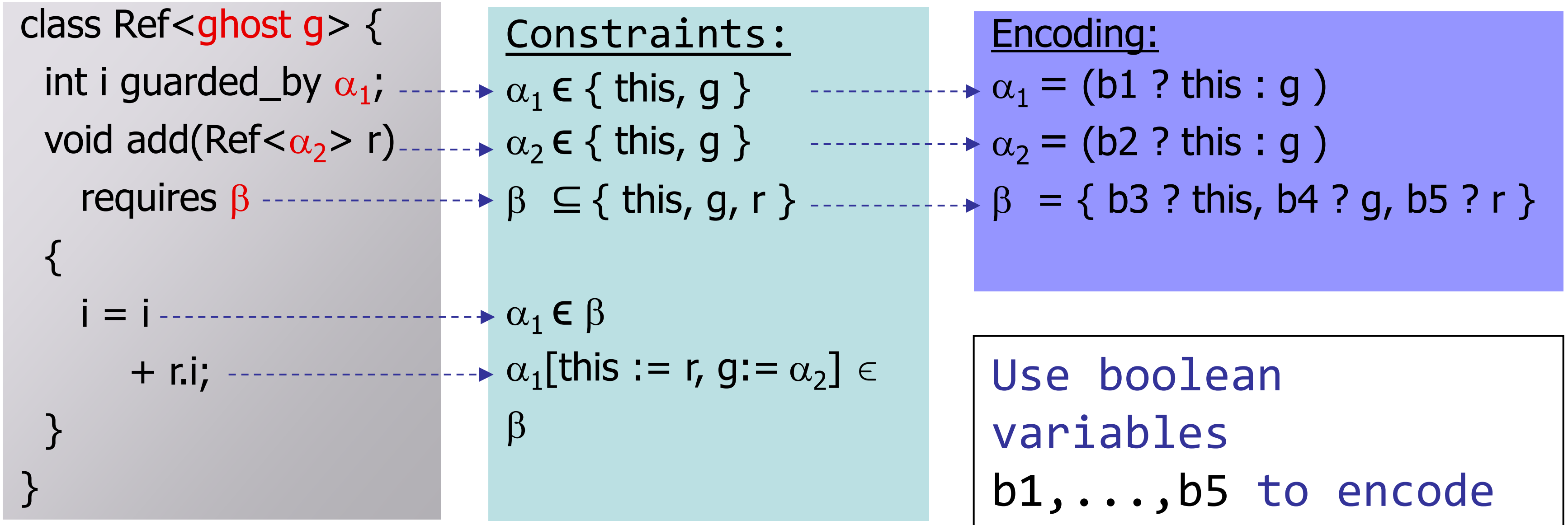
$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Reducing Type Inference to SAT



class Ref<ghost g> {
  int i guarded_by α_1;
  void add(Ref<α_2> r)
    requires β
  {
    i = i
      + r.i;
  }
}

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Encoding:

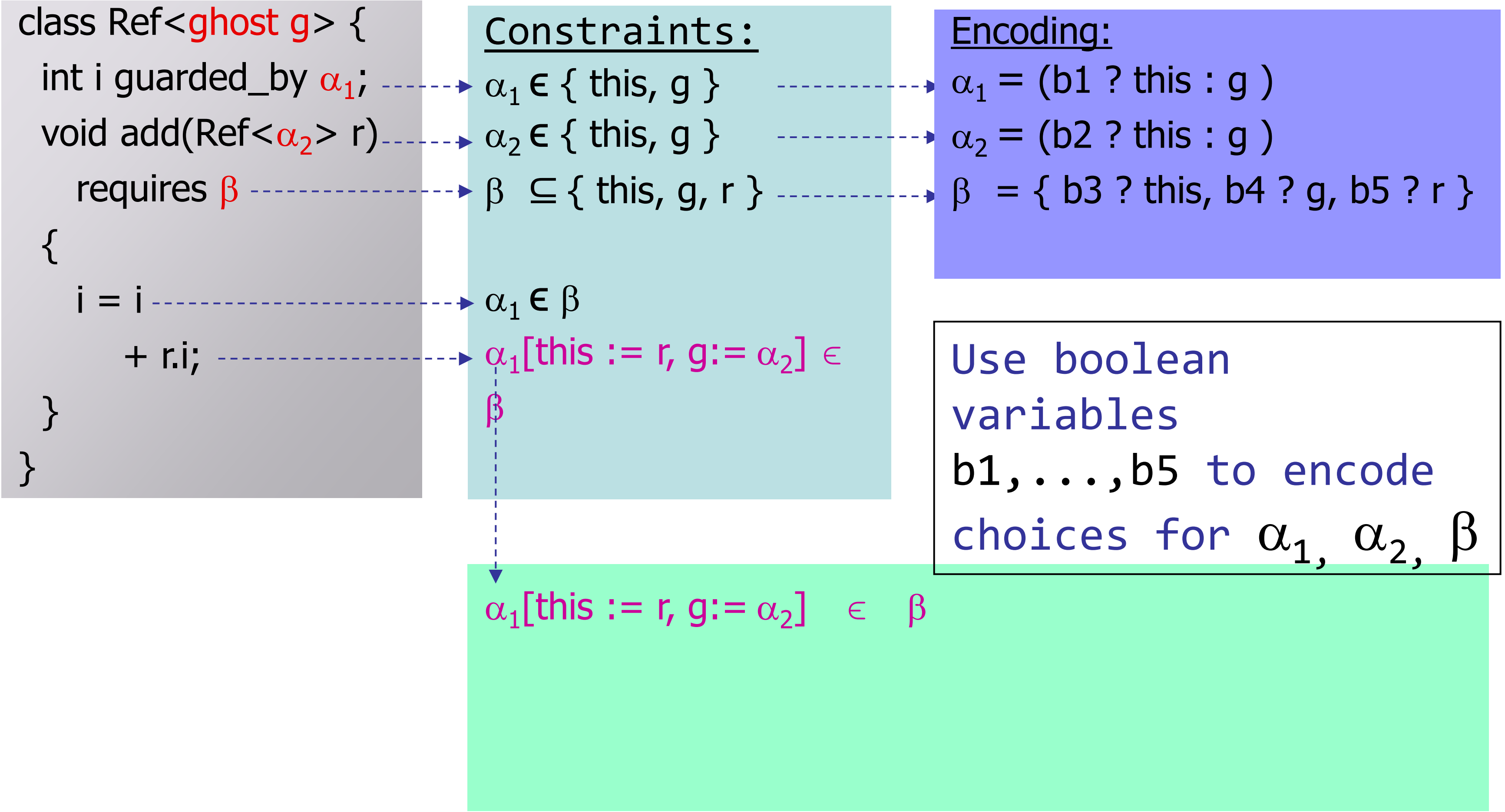
$\alpha_1 = (b1 ? \text{this} : g)$

$\alpha_2 = (b2 ? \text{this} : g)$

$\beta = \{ b3 ? \text{this}, b4 ? g, b5 ? r \}$

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by $\alpha_1$;
  void add(Ref<$\alpha_2$> r)
    requires $\beta$
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ this, g \}$

$\alpha_2 \in \{ this, g \}$

$\beta \subseteq \{ this, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[this := r, g := \alpha_2] \in \beta$

Encoding:

$\alpha_1 = (b1 \ ?\ this : g)$

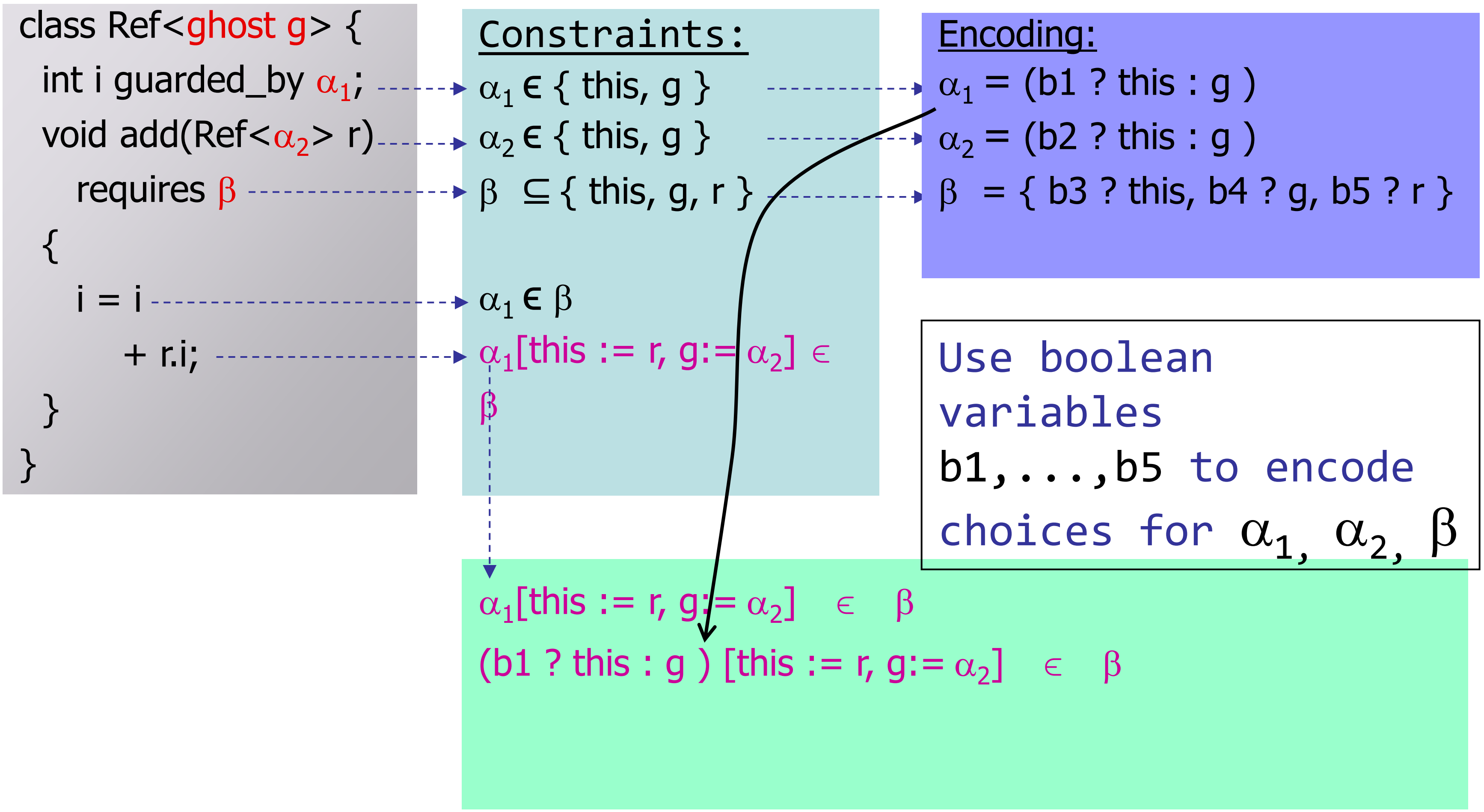
$\alpha_2 = (b2 \ ?\ this : g)$

$\beta = \{ b3 \ ?\ this, b4 \ ?\ g, b5 \ ?\ r \}$

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

$\alpha_1[this := r, g := \alpha_2] \in \beta$

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

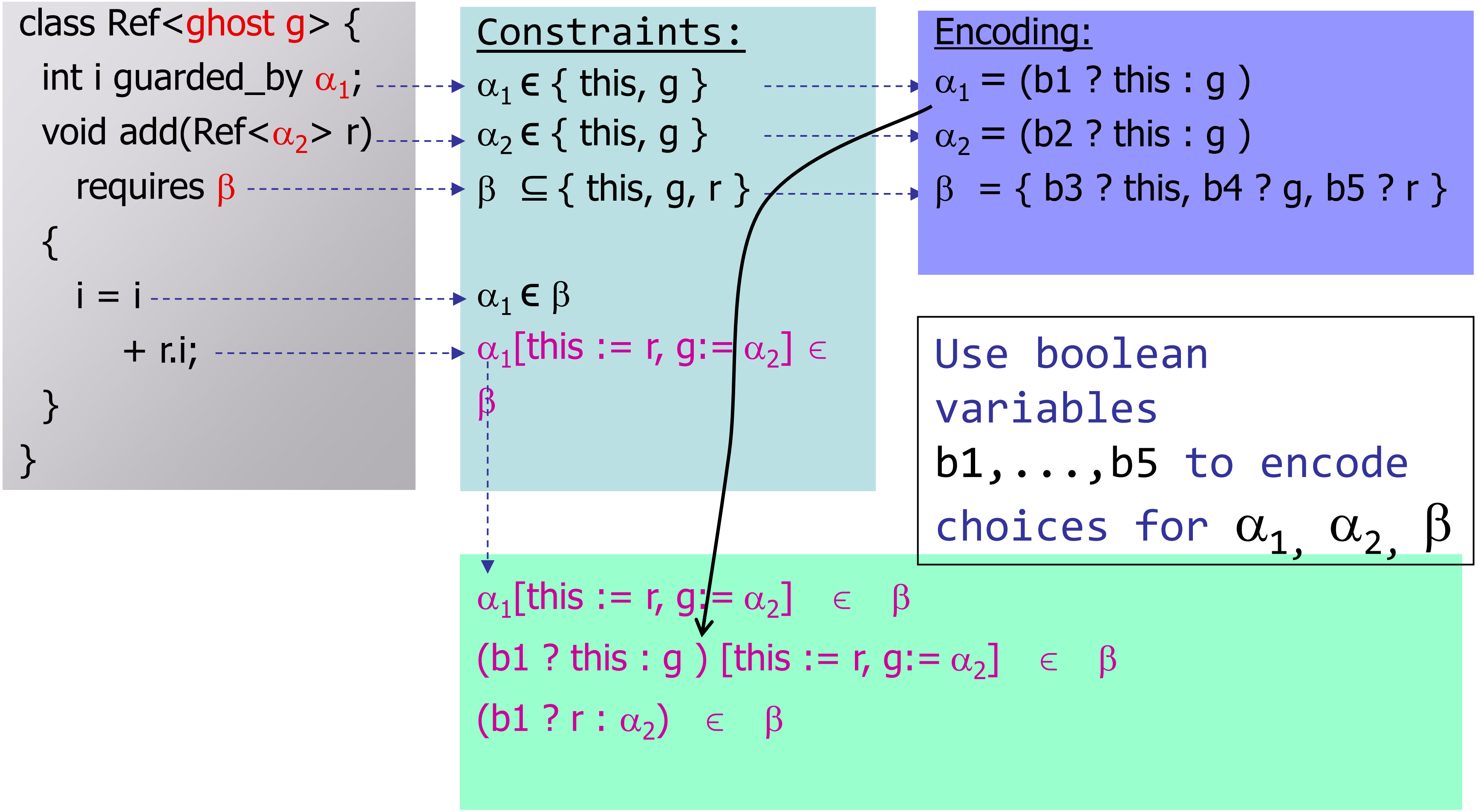
$\beta = \{ b3 \ ? \ \text{this}, b4 \ ? \ g, b5 \ ? \ r \}$

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ \text{this} : g)[\text{this} := r, g := \alpha_2] \in \beta$

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

$\beta = \{ b3 \ ? \ \text{this}, b4 \ ? \ g, b5 \ ? \ r \}$

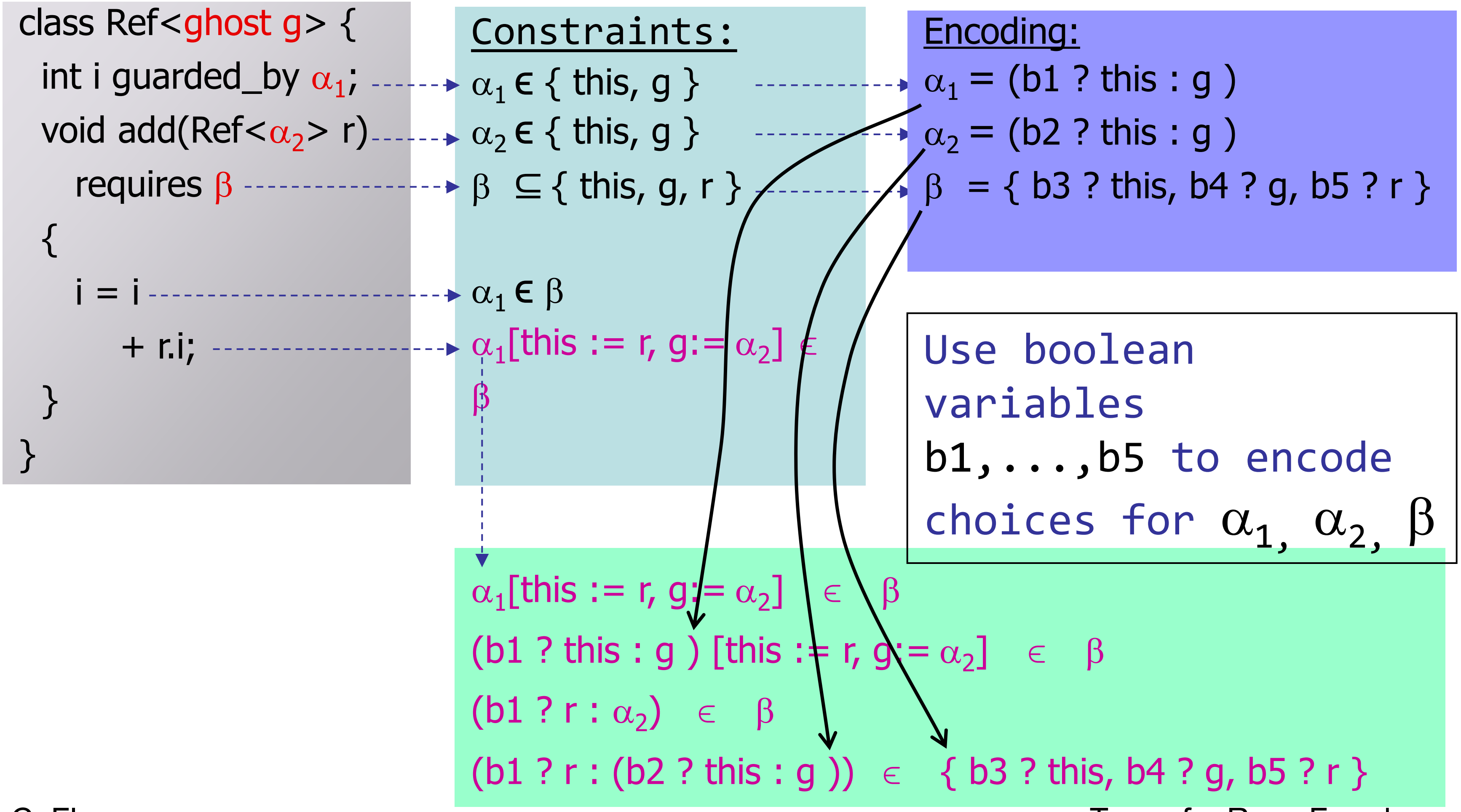
Use boolean variables b1,...,b5 to encode choices for $\alpha_1, \alpha_2, \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ \text{this} : g)[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ r : \alpha_2) \in \beta$

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Encoding:

$\alpha_1 = (b1 \; ? \; \text{this} : g)$

$\alpha_2 = (b2 \; ? \; \text{this} : g)$

$\beta = \{ b3 \; ? \; \text{this}, b4 \; ? \; g, b5 \; ? \; r \}$

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

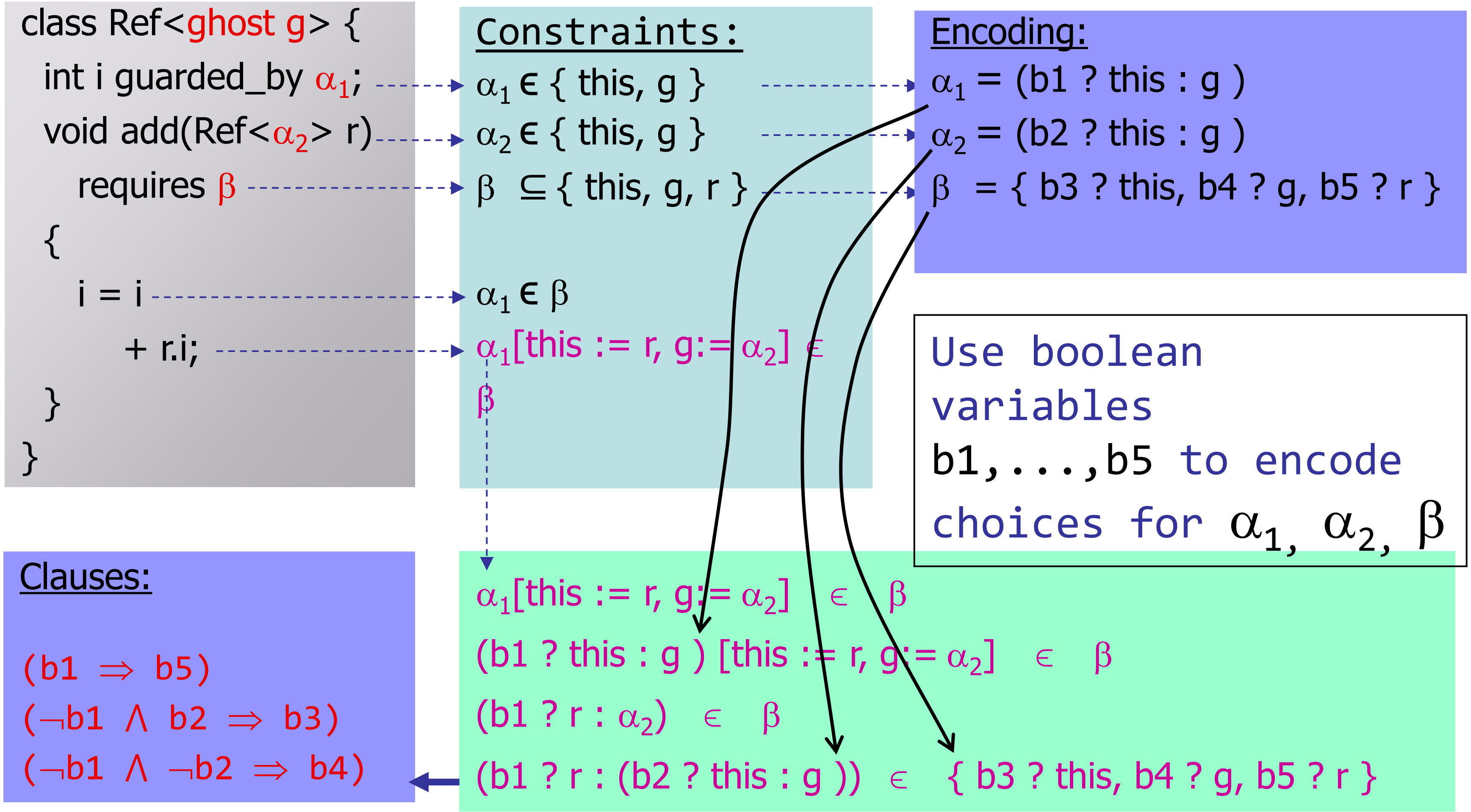
$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \; ? \; \text{this} : g)[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \; ? \; r : \alpha_2) \in \beta$

$(b1 \; ? \; r : (b2 \; ? \; \text{this} : g)) \in \{ b3 \; ? \; \text{this}, b4 \; ? \; g, b5 \; ? \; r \}$

Reducing Type Inference to SAT



```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
       + r.i;
  }
}
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

$\alpha_2 \in \{ \text{this}, g \}$

$\beta \subseteq \{ \text{this}, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

$\beta = \{ b3 \ ? \ \text{this}, b4 \ ? \ g, b5 \ ? \ r \}$

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

$\alpha_1[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ \text{this} : g)[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ r : \alpha_2) \in \beta$

$(b1 \ ? \ r : (b2 \ ? \ \text{this} : g)) \in \{ b3 \ ? \ \text{this}, b4 \ ? \ g, b5 \ ? \ r \}$

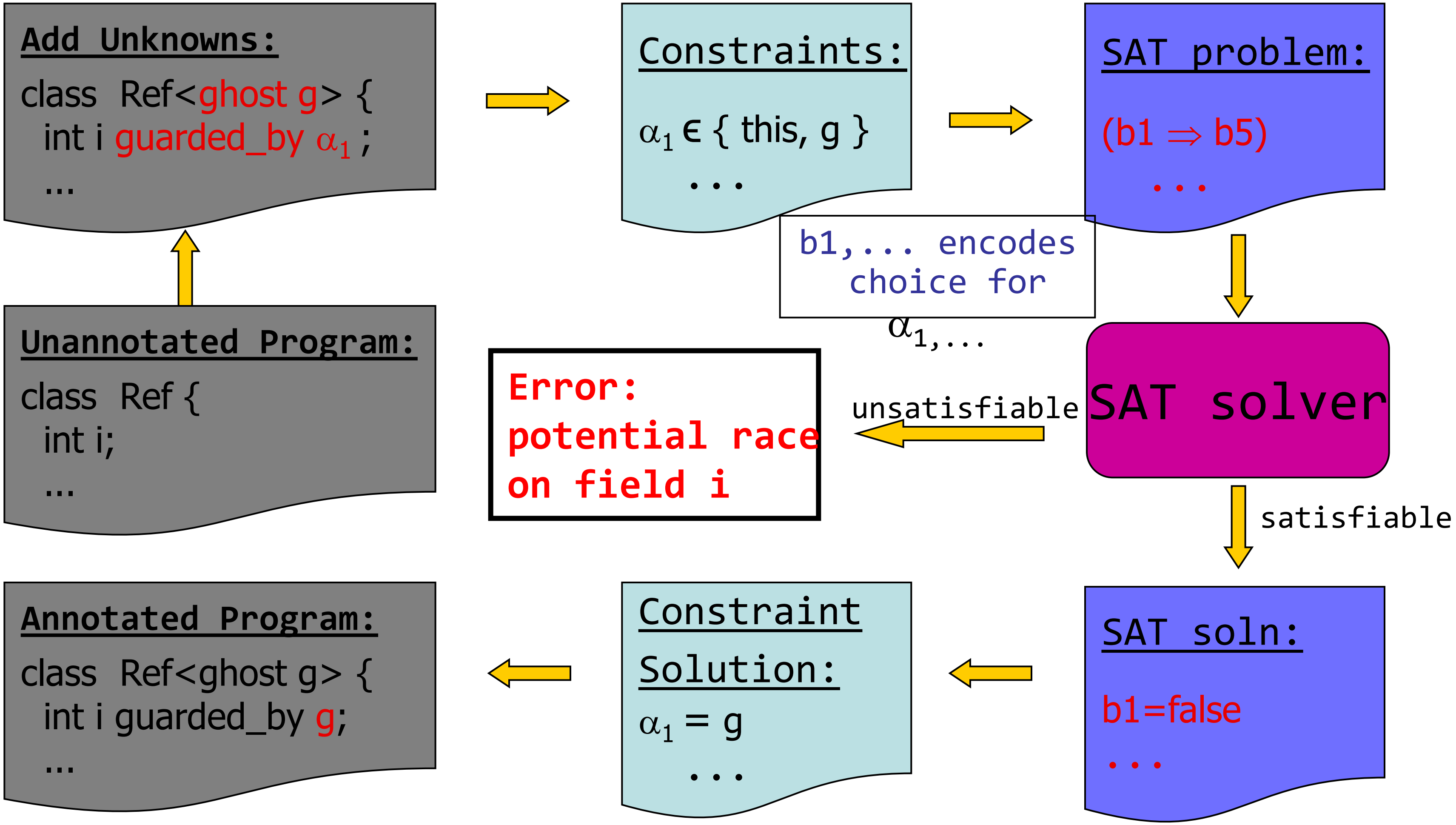
Clauses:

$(b1 \Rightarrow b5)$

$(\neg b1 \land b2 \Rightarrow b3)$

$(\neg b1 \land \neg b2 \Rightarrow b4)$

Overview of Type Inference



Unannotated Program:

```
class  Ref {
  int i;
  ...
```

Add Unknowns:

```
class  Ref<ghost g> {
  int i guarded_by α1 ;
  ...
```

Constraints:

$\alpha_1 \in \{ \text{this}, g \}$

...

SAT problem:

$(b1 \Rightarrow b5)$

...

b1,... encodes choice for $\alpha_{1},...$

SAT solver

unsatisfiable

Error: potential race on field i

satisfiable

SAT soln:

b1=false

...

Constraint Solution:

$\alpha_1 = g$

...

Annotated Program:

```
class  Ref<ghost g> {
  int i guarded_by g;
  ...
```